2002

Has the Object-Oriented Paradigm Kept Its Promise?

David Fernandez
Grand Valley State University

Al Fischer
Grand Valley State University

Michael Greco
Grand Valley State University

Bradly Hussey
Grand Valley State University

Steven Kuchta
Grand Valley State University

See next page for additional authors

Authors
David Fernandez, Al Fischer, Michael Greco, Bradly Hussey, Steven Kuchta, Hong Li, Steven Overkamp, Douglas Rodenberger, and Richard VanderWal

Has the Object-Oriented Paradigm Kept Its Promise?

Dr. Paul Jorgensen

David Fernandez
Al Fischer
Michael Greco
Bradly Hussey
Steven Kuchta
Hong Li
Steven Overkamp
Douglas Rodenberger
Richard VanderWal

9 December, 2002

Department of Computer Science and Information Systems
Grand Valley State University
1 Campus Drive
Allendale, MI 49401
USA

Table of Contents:

I Introduction

II. History of the Object Oriented Paradigm
- A. Background
  - 1. What is Object-Orientation?
  - 2. History of OO
- B. Promises of Object-Oriented Programming (OOP)
  - 1. Promises
  - 2. Other Promises (or Marketing Hype) of OOP
- C. Discussion of why OO took so long to gain popularity
  - 1. The Year of the Object
  - 2. Advancement of OO technology and use
- D. Summary

III. What is Pure Object-Oriented?
- A. Introduction
- B. Abstraction
- C. Encapsulation
- D. Hierarchy
- E. Modularity
- F. Typing
- G. Software Development
- H. Language Comparisons

IV. Complexity
- A. Introduction
- B. Comparison with Complexity Metrics
- C. Commercial Training
- D. University Education
- E. Conclusion

V. Naturalness
- A. Natural Conceptuality and Abstraction
- B. Abstraction and Objects
- C. The traditional alternative- the Procedural paradigm
- D. Focus on Communication
  - 1. Accuracy of Communication – Understanding each other
  - 2. A Good Tool – to the rescue
- E. Realizations
- F. Conclusion

VI. Powerful Modeling
- A. Introduction
- B. Promise

C. Actuality
D. Conclusion

VII. Object Oriented Software Development and Extendibility
A. Introduction
B. Extendibility Defined
C. Common Inheritance Models
D. Subclassing for Specialization
E. Subclassing for Specification
F. Subclassing for Construction
G. Subclassing for Generalization
H. Subclassing for Extension
I. Subclassing for Limitation
J. Subclassing for Variance
K. Subclassing for Combination
L. Extendibility Conclusion

VIII. Reuse
A. Introduction
B. Hopes for Reuse
1. Reduced Coupling
2. Complexity Hiding
3. Increased Scalability
4. Better Division of Work
5. Better Abstraction
6. Improved time to market
C. Obstacles to Reuse
1. To Reuse or not to Reuse, that is the Question
2. Active Management of Reuse Strategy
3. Economical Challenges
4. Administrative Impediments – Cataloging, Search and Discovery Limitations
5. Problem Reporting and Version Control
6. Narrow Focus on Code Reuse Only
7. Getting started is Difficult
8. Opportunistic Reuse – Convenience and Safety vs. Proactive Design
9. Reorganization to Escape Silo-Oriented Development
10. Psychological Impediments
11. Reuse Measurement & Incentives
12. Surface vs. Structural Similarity – Expertise Level
13. Expressing Management's Commitment to Reuse Strategy & Accountability for ROI
14. Reuse Metrics
15. Knowing When to Outsource
16. Skill Levels Required of OO Analysis & Development

D. Design Patterns Bridge the Technical Gap
E. Successful Implementation Examples
F. Conclusion on Reuse

IX. Testing and Debugging Object Oriented Systems
A. Introduction
B. The Testing Process
1. Functional and Structural Testing
2. Levels of Testing
3. Definition of a Bug
C. Hopes for Easier Testing
1. Reuse
2. Iterative Development
3. Encapsulation
D. Obstacles to Testing
1. Decentralized Code
2. Exceptions
3. Test Case Identification
4. Encapsulation
5. Inheritance
6. Polymorphism
E. Conclusion

X. Reduced Maintenance
A. Introduction
B. Definitions
C. Accountability
D. Encapsulation vs. Complexity
E. Documentation
F. Modularity and Testing
G. Modularity and Reuse
H. Proper Use of OO Concepts
I. Learning Curves, Specialization, and Salary
J. Inheritance
K. Conclusion

XI. Conclusion

XII. Works Cited

Appendix A – Author Information

I. Introduction

This report is the conclusion of the Fall 2002 Capstone Seminar for the Master's Degree in Computer Information Systems program at Grand Valley State University. The seminar had nine participants who, together with the instructor, spent the semester examining object-oriented software development. As a group, the seminar participants represent 103 years of industrial software development and management experience. Each participant provided a brief self-description that is located in Appendix A.

The Capstone Seminar is an integrative course in which students lead a discussion and present a paper on a current topic (in this case, some aspect of object-oriented software development). In addition, the Capstone Seminar promotes collaboration among graduate students and gives both faculty and graduate students an opportunity for sustained work on topics outside the scope of regular course offerings.

At the beginning of the semester, our plan was to identify the hopes promised by the early writers on object-oriented software development, and then study the extent to which these hopes have been realized. There was some discussion of possible titles for this report. Some of the contending possibilities were:

- Object-Oriented Software Development: Hype or Hope?
- Is Object-Oriented Software Development the Silver Bullet?
- Broken Promises of Object-Oriented Software Development
- Smoke, Mirrors, and Object-Oriented Software Development

While these captured some of the class sentiment, in the end, we stayed with the more neutral one. As a group, the seminar participants are mildly divided on Object-Oriented Software Development: there are skeptics, enthusiasts, and some who remain undecided. In our examination of the various issues, we observed many points at which the various issues intersect. To highlight these, we have organized this report with internal links; it is best read online. Our report represents forty hours of group meeting and discussion time, based on nearly 700 hours of individual preparation time.

Our first major observation was that there are two categories of object-oriented promise: management oriented and technology oriented. The management oriented promises are easy–faster, better, cheaper, and more software. The technology oriented promises are well known: reduced complexity, "natural" thought processes that lead to more powerful modeling, software that is easy to extend (and hence, to maintain), increased potential for re-use, easier software testing, and finally, easier maintenance. The incidence between these two views of the promise of object-oriented software development is shown in the table below.

	Faster	*Better*	*Cheaper*	*More*
Complexity	√	√	√	√
Naturalness	√	√	√	√
Modeling	√	√		√
Extensibility		√		√
Reuse	√	√	√	√
Easier testing	√	√	√	
Maintenance		√	√	

Each of the technical hopes is expanded into a more complete discussion. These sections follow two background discussions on the history and our working definition of "pure" object-orientation.

II. History of the Object Oriented Paradigm

A. Background

1. What is Object-Orientation?

Early computer languages could be identified as being procedural and applicative in nature. They were a merely a series of instructions that assigned a value to a variable. Rather than separating procedures and data into separate components, Object-Oriented (OO) languages combined them into a collection of subroutines packaged together, along with the data it uses, called a class. A particular instance of a class is called an object. Other qualities of OO include inheritance, encapsulation, and polymorphism. Inheritance, the major distinguishing feature of object-oriented programming, is a method that OO programmers can use to share, or reuse, common parts of a class. Encapsulation is the process of placing elements of a program into a central location, grouped by topic, and then developing functions to access and perform operations on those topics. Polymorphism refers to a programming language's ability to process objects differently, depending on its data type or class. Specifically, it is the ability to redefine methods for derived classes.

2. History of OO

Although the concept that everything is an object has been around since the beginning of man, the concept of software objects didn't start to take shape until the early 1950s. There are, in fact, many definitions of an object, such as the one by Grady Booch [Booch, p77]: "An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable".

OOs early growth was inhibited by a number of factors, which, among other reasons included: lack of technology, high cost of computer memory and hardware, and a great reluctance to change technologies, by management. So how did OO begin?

"The term object was first formally applied in the SIMULA language, and objects typically existed in SIMULA programs to simulate some aspect of reality" [Booch, p77].

Kristen Nygaard is credited with the development of SIMULA, the first software application to use OO, while doing work in Operational Research in the 1950s and early 1960s at the Norwegian Computing Center, Oslo, Norway. The research that he was doing there created the need for precise tools so he could describe and simulate complex man-machine systems. One of the major problems Nygaard encountered in this research project was how to describe the heterogeneity of a system and its operation. In the 1950s, modeling of such systems was usually done through means of symbol notation, i.e. flow-diagrams accompanied by an account of the rules governing the operations of the system.

In 1961, Nygaard was working with a programmer named Ole-Johan Dahl that he had previously worked with. Shortly after they got together again, the idea emerged that they needed to develop a language which contained an algorithmic language that could be used both for system description (for people) and for system prescription (as a computer program through a compiler). SIMULA developed from this work between 1962 and 1965. In 1967 they added the inheritance mechanism. SIMULA was originally designed and implemented as a language to simulate discrete events, but was later upgraded and re-released as a full scale programming language. Although SIMULA never became widely used, the features used by it were highly influential on modern programming methodology. Among other things, SIMULA introduced the object-oriented programming concepts like classes, objects, inheritance, encapsulation and polymorphism, as were described above.

In 1972, a pure OO programming language, Smalltalk, was created by Alan Kay at Xerox PARC. Smalltalk used the concepts of classes and messages, that were originally introduced by the SIMULA language, to further promote the idea of using software objects to model real world objects.

Since 1972, literally hundreds of OO languages have been developed and released along with thousands, if not tens of thousands of OO programs. Most, if not all, of them were introduced to the market place with some basic promises, and lots of hype. Only a few of them have had any real success. The top four OO languages in use today include Smalltalk, Eiffel, C++ and Java.

B. Promises of Object-Oriented Programming (OOP)

1. Promises

"Object-orientation is a new technology based on objects and classes. It presently represents the best methodological framework for software engineering and its pragmatics provides the foundation for a systematic engineering discipline. By providing first class support for the objects and classes of objects of an application domain, the object-oriented paradigm precepts better modeling and implementation of systems. Objects provide a canonical focus throughout analysis, design, and implementation by emphasizing the state, behavior, and interaction of objects in its models, providing the desirable property of seamlessness between activities." (Robert John Hathaway)

"Object-oriented languages and systems are a developing technology. There can be no agreement on the set of features and mechanisms that belong in an object-oriented language since the paradigm is far too general to be tied down. We can expect to see new ideas in object-oriented systems for many years to come." (Oscar Nierstrasz)

(Since Nierstrasz's time, it has been generally been accepted that a pure OO language has 5 general characteristics. These characteristics are discussed in Section III.)

These modest promises made by Hathaway and Nierstrasz regarding the future of OO, the developers and, perhaps even more so, the marketers, of OO languages soon gave way to bolder and grander promises. These promises, which were intended to further separate OO languages from the more traditional languages included:

- Naturalness - OOP models the real world better because everything in the world is an object.
- Reuse - OOP makes programming faster and easier because of the reuse of existing, previously tested classes from a library. Reuse can also be accomplished by inheritance.
- Development Life Cycle - OOP makes faster development of programs because rapid prototyping of models is achieved due to the reuse of existing models of a corporations processes.
- Maintenance - OOP makes maintenance easier because code only needs to be changed in one place. This is made possible by encapsulation.
- Quality - OOP makes testing easier and more reliable because components are existing and previously tested.

2. *Other Promises (or Marketing Hype) of OOP*

The advertising people also became involved in marketing OO. Some of the "hype" associated with OO programs included:

- OOP is a proven general-purpose technique - OOP has been in use for many years and improves software design and performance.
- OOP does automatic garbage-collection better - Automatic recovery, or de-allocation, of memory by destructors.
- OOP divides up work better
- OOP "hides complexity" better
- OOP eliminates the "complexity" of "case" or "switch" statements
- OOP reduces "coupling"
- OOP makes programming more visual - GUI Interfaces are easier to program.
- OO can "protect data" better - OOP hides data from the public by encapsulation.
- OO is scaleable - OOP makes larger systems easier to build and maintain because subsystems can be developed and tested independently.
- OO makes you a better programmer
- OO software is superior to other forms of software
- OO techniques mean you don't need to worry about business plans
- OO will cure your corporate ills

C. Discussion of why OO took so long to gain popularity

1. *The Year of the Object*

In a commentary regarding an article titled, "The Year of the Object", published in the August 1989 issue of Computer Language Magazine, Ed Yourdon (Yourdon) states that,

" It appears that 1989 will go down in the computer industry's history books as the Year Of The Object. You can't pick up a book or magazine today, nor can you attend a conference or symposium, without being inundated with object-oriented this or object-oriented that. Object-oriented programming has been around for a long time, of course, and now object-oriented design has achieved a lot of credibility; some people (modesty prevents me from naming names) have even documented an object-oriented approach to systems analysis."

2. Advancement of OO technology and use

Why did it take several decades after OO was developed and marketed in the late '60s in order for "The Year of the Object" to come about? A look back at the history of the computer industry, in general, may explain what influenced, or impeded, the development of OO technology and use.

Up until the 1940s "computers" used punched card inputs and if not for World War II it is possible that they still could today. However, many programmable computers such as the Mark I, ENIAC, EDVAC, Whirlwind and Ferranti Mark I, were invented at universities. With funding coming mostly from the Defense Department to support the war effort. Other countries were also funding computer research. For example, Konrad Zuse, a German engineer, completed the first general purpose programmable calculator in 1941. Also, Colossus, a British computer developed for code-breaking, became operational in December of 1943.

Bell Telephone Laboratories developed the transistor in 1948, which would eventually replace the vacuum tubes used by the first generation of computers It took many years of work to overcome production problems. It wasn't until 1959 that the second generation of computers were born.

All second generation computers used core memory. Core memory consisted of tiny iron "donuts" strung on a grid of wires. Core memory was first used in 1953 and was used until RAM chips replaced it in the early 1980s. Core memory required, depending on the model, 4.5 to 11.5 microseconds per character. Each character consisted of 6 bits and the number of characters in memory ranged from 1.4K up to 80K. Depending on the amount of memory installed, it could cost between $4,000 to $18,000 per month to rent an IBM 2nd generation computer.

OO programs tend to run slower than non-OO programs. This is true because when an OO program is running, the system has to do a lot more work per function, and OO programs tend to need more memory. Data storage was very expensive and class libraries can become very large, requiring huge amounts of storage. Nygaard realized in 1960 that neither of the first generation computers, a Ferranti MERCURY and a DEUCE , being used at the Norwegian Computing Centre (NCC) would be able to handle the computing requirements of his new program. Shortly after accepting the fact that the NCC would need to settle for a GIER computer, Nygaard was invited by the Univac Division of the Sperry Rand Corporation to see their new UNIVAC 1107 computer. After much negotiating he was able to make a very good deal with Sperry Rand that would allow the NCC to purchase a new UNIVAC 1107, which was considerably faster than any computer existing at the time.

The third generation of computers occurred in 1964 and lasted to 1967. It was marked by the production of the first modern computer families such as the IBM 360, the CDC 6600 and DECs PDP-6, PDP-8 and PDP-10. Several new languages were developed during this time, mostly for use on the new computers. Most IT managers, however, decided to stay with the old legacy programs (i.e. COBOL and FORTRAN). Switching to a new language would have required too much time, and money, to teach the new languages to programmers and to rewrite all of the existing software. This was a philosophy that would also slow down the popularity and, hence, the use of OO programs for many years to come, and perhaps even still today.

In the early 1970s, SIMULA was upgraded several times to add new features and to operate more efficiently on third generation computers. The DEC System-10 SIMULA, for example, was revised specifically for DECs PDP-10 computer. It was in many ways more comprehensive then it's predecessors. It contained, among other things, online debugging facilities which allowed setting and re-setting of break points during program execution. The DEC System-10 SIMULA compiler was especially designed for interactive use, and would soon set a new standard for the development of other SIMULA compilers

In the meantime, memory chips became cheaper than core memory , and many magnitudes faster. Finally, in 1975, the first integrated circuit, a complete CPU on a chip, was delivered. As the price of computer memory dropped, the interest in OO began to increase again resulting in 1986 becoming truly, "The Year of the Object".

D. Summary

It has been claimed by many people that OO is a powerful tool whose time has not yet come.   One reason is that its use is not right for all applications, or for all industries. However, as the price of computer memory comes down, the speed of the CPUs increases, and managers get more experience with OO programs, OO programming may well catch up with the promises, and perhaps even most of the hype, of the old days.

III. What is Pure Object-Oriented

A. Introduction

It is the goal of this section to identify and define the fundamental concepts that distinguish object-orientation from other software development paradigms. Once these fundamentals are understood, this knowledge should identify why the proponents of object-orientation had the hopes they did and why they believed this is a better approach to software development. In contrast, given the state of software development today, these fundamentals should also help identify why these goals may not have been realized, as many assumed they would automatically be achieved by adoption of object-orientation. In order to define what "pure" objected-oriented means, it must first be defined what an object is, then what is meant by orientation, and finally how all this relates to the world of software development.

The world is made up of things. The important part of what we see is that each of these things has certain characteristics or properties and it exhibits some types of behavior that makes it unique. These characteristics and behaviors are tightly coupled in a singular relationship that provides a mechanism for classifying that thing into a unique type. These things are objects! IEEE defines an object as "an encapsulation of data and services that manipulate that data." [IEEE Std 610.1211990] An orientation gives us a mechanism by which we view things. Putting the two terms together then, "object-orientation" simply means that the method of world observation is in terms of objects, identifying things by their characteristics and behavior. These definitions are rather simplistic and don't provide any concise concepts that make this paradigm any more detailed.

How do the promoters of object-orientation define this concept? "The basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance and a mechanism that allows calls of member functions to depend on the actual type of an object (in cases where the actual type is unknown at compile time)." (Stroustrup 1991). While this description is a little less fuzzy, it still seems to be incomplete if we don't understand a class mechanism, inheritance, and member functions. In Booch (1994), he identifies the existence of 7 elements in the object model, 4 major and 3 minor. The major elements are Abstraction, Encapsulation, Modularity, and Hierarchy and the minor elements are Typing, Concurrency, and Persistence. He states, "A language can only be considered object-oriented if it supports all of the major elements". These two definitions fairly well summarize and make explicit the characteristics of object-orientation as detailed by many of the proponents. Tying these two definitions together, a set of 5 common characteristics of object-orientation can be deduced: Abstraction, Typing, Hierarchy, Encapsulation, and Modularity. Each term will now be defined to further the understanding of this paradigm.

B. Abstraction

The world of software development has increased in complexity at an astonishing rate. The level of application functionality desired by customers has grown phenomenally. Software applications today are doing everything from controlling nuclear power plants to managing and tracking billions of dollars in commerce. How is this complexity dealt with since a human can only comprehend and manage a limit amount of information at a time? As the title to this section alludes, it is done through a process of abstraction.

Abstraction is simply the ignoring of the irrelevant pieces and focusing on the pieces that are important. In software, a representation of some thing is created. It is not the real thing and it is much less complex by focusing only on the attributes and behaviors that are important to the application. In object-orientation, the term "class" or "type" is commonly used to describe a unique category of things, while the term "object" is used to denote an instance of a particular class. An example of this is the class "car". This category would be defined by the attributes (color, number of passengers, number of wheels, engine size) and the behaviors (go, stop, breakdown) that are thought to be important. Attributes and behaviors are allocated to the abstraction in terms of the level they exit at. Most attributes are instance attributes because they can change from one instance to another. Color is a good example of an instance attribute because you see cars in many different colors. If it were assumed that all cars have only 4 wheels, then this may be deemed a class attribute. All car instances contain the same value for this attribute. If the value changes, it changes for all instances. This same allocation between class level and instance level can be made to class behaviors also. If an instance method is performed, that behavior only occurs to the single object, whereas if a class method is executed, the behavior is performed for all object instances. Let's be glad that breakdown isn't a class behavior, it already takes forever to get your car fixed and back from the mechanic.

C. Encapsulation

Encapsulation is also commonly referred to as "Information Hiding". In the previous section we talked about how the attributes and behaviors of a class distinguish it from all other classes. Encapsulation deals with permitting or restricting a client class' ability to modify the attributes or invoke the methods of the class or object of concern. If a class allows another to modify its attributes or invoke its methods, the attributes and methods are said to be part of the class' public interface. If a class doesn't allow another to modify its attributes or invoke its methods, those are part of the class' private implementation. A "Queue" provides a good example of this characteristic. A queue is an abstract concept that represents an ordered list of things. The implementation of a queue may by an array or it may be by a linked-list. If the implementation were known, a developer writing a client of the queue class may use this knowledge and directly access the internal storage mechanism. If the implementation changed, the client would then have to be modified also. This type of tight coupling between classes would cause a very brittle system and would increase maintenance costs as parts of the system were modified. Therefore, the levels of encapsulation that a language supports and how those mechanisms are used directly impacts the level of coupling between classes and it can significantly affect the cost of maintenance in an application.

D. Hierarchy

In the pursuit of defining abstractions, it is realized that some types have many of the same attributes and/or behaviors as other classes. Classification in most scientific exploits commonly deals with one type being a specialization or generalization of another. A dog is an animal; a cat is an animal. There are different types of dogs and cats. If the process of classification normally exhibits a hierarchy, shouldn't the abstractions used in software development also exhibit and benefit from this feature. Hierarchy is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way. This hierarchy is implemented in software via a mechanism called "Inheritance". Just as a child inherits genes from its parent, a class can inherit attributes and behaviors from its parent. The parent class is common referred to as the super-class and the child class as the sub-class.

Another implementation feature of the hierarchy characteristic is referred to as "Generics or Generic Units". An example best serves to illustrate this concept. Using the idea of a "Queue" class as described above. It wasn't mentioned the type of objects that the queue held. If the current project needed a queue of integers, the queue would be implemented with that type. What would happen if the next application were a simulation of a line of customers for a bank teller? The queue class would have to be re-written, this time with the Customer type instead of an integer. Obviously having to re-write this abstraction with all of the different permutations of contained items is not in the best interest.

E. Modularity

Modularity is probably is the most common term of everyday language used in defining the characteristics of object-orientation. Webster's dictionary defines modular as "designating or of units of standardized size, design, etc. that can be arranged or fitted together in a variety of ways." Even Booch defines modularity as "the property of a system that has been decomposed into a set of cohesive and loosely coupled modules." Modularity simply means the physical partitioning of the application where as abstraction is the logical partitioning. A module can be considered a class, a file, a package, or almost anything. The issue with these definitions is that while they explain what modularity is, they don't explain how to achieve the qualities of "cohesive" or "loosely coupled". The quality of a module is completely dependent upon the ability of the person constructing it. Bertrand Meyer has probably addressed the term modularity and the development of modules to the most extent of anyone. He defines five criteria of modularity as Decomposability, Compos-ability, Understandability, Continuity, and Protection. Therefore, a key to how well modularity can be achieved is how restrictive or permissive the definition of a module is, in a language implementation. A module is the primary mechanism for reuse.

F. Typing

Typing commonly refers to how stringent a compiler is when determining whether or not a type has a specific attribute or operation when called by the source code. For example, our “Car” class has the operation “go” but does not have an operation “speed”. If a software developer attempted calling “speed” on a car instance, the style of typing determines where this error would be caught. Strong typing means that the error would be caught at compilation time while weak typing means that the error would be caught at runtime.

Typing is also associated with characteristic of hierarchy. If a super-class defines a public method, a sub-class has the ability to redefine that method in terms of the behavior desired in the sub-class. An example of this is a class “Animal” with a method called “speak” in its public interface. There are two sub-classes of “Animal”, “Dog” and “Cat”. Class “Dog” may override the “speak” method to produce a barking sound while class “Cat” will override the method to produce a meow sound. Polymorphism is the ability of two classes to react differently to the same message.

G. Software Development

Software development is similar to most other types of construction processes. A complex problem is encountered; a solution is deduced; then construction of the solution occurs. When performing these steps in software development, we commonly refer to them as analysis, design, and implementation. “Object-orientation” in a software development process then means we focus on types, characteristics, and behavior as we perform each of the steps, object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP). The fact that we tie each of these steps together in an object-oriented fashion, that the solution domain is expressed in the same types as the problem domain, is referred to as seamless-ness. The existence of the object-oriented characteristics in a software development language has a significant impact on the design process, as the design is "the disciplined approach we use to invent a solution for some problem." If the implementation language lacked a specific characteristic, specifying or making use of that characteristic in design would make very little sense.

As detailed in the introduction to this document, proponents expressed numerous promises to the adopters of the object-oriented paradigm. The incidence matrix below shows the relationship between the promises made and the characteristics of object-orientation. An "X" mark indicates a correlation between the existence of a characteristic and the ability to realize a promise. This chart will also help show how the selection of an object-oriented programming language may impact which promises may be realized as each language supports to a varying degree a sub-set of the characteristics of object-orientation.

	Complexity	Naturalness	Reuse	Extensibility	Easier Testing	Less Maintenance	More Powerful Modeling
Abstraction	X	X	X	X			X
Typing				X	X	X	
Encapsulation	X				X	X	
Hierarchy	X	X	X	X			X
Modularity			X		X	X	

H. Language Comparisons

The following language comparisons have been adapted from the comparisons that Booch (1994) created in his book. They have been modified to incorporate the characteristics identified above and updated to reflect the current state of popular object oriented languages.

Ada95		
Abstraction	Instance variables Instance methods Class variables Class methods	Yes Yes No No
Encapsulation	Of variables Of methods	Public, Private Public, Private
Modularity	Kinds of modules	Package
Hierarchy	Inheritance Generic Units	Yes Yes
Typing	Strongly typed Polymorphism	Yes Yes

C++		
Abstraction	Instance variables Instance methods Class variables Class methods	Yes Yes Yes Yes
Encapsulation	Of variables Of methods	Public, Protected, Private Public, Protected, Private
Modularity	Kinds of modules	File
Hierarchy	Inheritance Generic Units	Yes (multiple) Yes
Typing	Strongly typed Polymorphism	Yes Yes (single)

C#		
Abstraction	Instance variables Instance methods Class variables Class methods	Yes Yes Yes Yes
Encapsulation	Of variables Of methods	Public, Protected, Private Public, Protected, Private
Modularity	Kinds of modules	File
Hierarchy	Inheritance Generic Units	Yes (single + interfaces) No
Typing	Strongly typed Polymorphism	Yes Yes (single)

Eiffel		
Abstraction	Instance variables Instance methods Class variables Class methods	Yes Yes No No
Encapsulation	Of variables Of methods	Private Public, Private
Modularity	Kinds of modules	Unit
Hierarchy	Inheritance Generic Units	Yes (multiple) Yes
Typing	Strongly typed Polymorphism	Yes Yes (single)

Java 1.4.1		
Abstraction	Instance variables Instance methods Class variables Class methods	Yes Yes Yes Yes
Encapsulation	Of variables Of methods	Public, Package, Protected, Private Public, Package, Protected, Private
Modularity	Kinds of modules	Class
Hierarchy	Inheritance Generic Units	Yes (single) No
Typing	Strongly typed Polymorphism	Yes Yes (single)

Smalltalk		
Abstraction	Instance variables Instance methods Class variables Class methods	Yes Yes Yes Yes
Encapsulation	Of variables Of methods	Private Public
Modularity	Kinds of modules	None
Hierarchy	Inheritance Generic Units	Single No
Typing	Strongly typed Polymorphism	No Yes (single)

IV. Complexity

A. Introduction

Designing software is a complex process. Complexity of object-oriented software is a major element of the cost, reliability, and functionality of software systems. Furthermore, complexity affects training time, reusability, portability and maintainability. Therefore, the complexity and training associated with object-oriented software is a big issue for object-oriented languages. Software design complexity is an important factor affecting the cost of software maintenance. The degree of complexity factors affects the cost of maintenance. If we can determine the impact of the complexity factors on maintenance effort, we can develop guidelines, which will help reduce the costs of maintenance by recognizing troublesome situation early. Complexity influences software reuse in three different ways. First, a component has to be understood before it is extracted, modified, and finally reused. Second, a class with a large number of methods is likely to be more application specific, which reduces its reusability (as discussed in the previous hypothesis). Third, a complicated code component also demands a greater level of understanding on the part of testing and debugging the class. Thus, the cost of using such components is high because high complexity and low readability make modification and integration more difficult. We examine the question of object-oriented software from two standpoints: directly with mainline software complexity metrics and indirectly in terms of university education and commercial training.

B. Comparison with Complexity Metrics

There are several ways to measure complexity in object-oriented software, this comparison considers non-comment Lines of Code, cyclomatic complexity, and Software Physics (Halstead),

(1) Cyclomatic complexity
Cyclomatic complexity measures the amount of decision logic in a single software module. It is based entirely on the structure of software's control flow graph, in which nodes correspond to statement fragments, and edges: represent transfer of control between nodes. Each possible execution path of a software module has a corresponding path from the entry to the exit node of the module's control flow graph.

(2). Halstead

The Halstead measures are based on four scalar numbers derived directly from program source code:

- N1 = the total number of operators..
- N2 = the total number of operands.
- n1 = the number of distinct operator occurrences
- n2 = the number of distinct operand occurrences

From these quantities, four measures are computed:

- Program length $N=n1 * \log_2 n1 + n2 * \log_2 n2$
- Program vocabulary $n=n1+n2$
- Volume $V=N*(LOG_2\ n)$
- Difficulty $D=(n1/2)*(N2/n2)$

Program vocabulary attempts to describe the information a programmer must maintain while coding. Halstead's term is "mental discriminations". Program length and volume are indicators of program size, which in turn, indicate complexity.

Metric	Procedural pseudo-code	O-O pseudo-code
non-comment lines of code	50	63
Cyclomatic complexity	12	30
Program vocabulary n	20	58
Program length N	69	298
Volume (V)	505	835
Difficulty (D)	13.8	51.3

From the above illustration, object-oriented software is much more complicated than the procedural method. Learning object-oriented software programming languages is similarly more difficult.

C. Commercial Training

The employment market reflects this the complexity of object-oriented software in a monetary way. Object-oriented developers' salaries are higher than those of legacy developers. In reading a paper, Java developer compensation averages $94,000 per year, up to 20 percent higher than that of legacy developers (Feiman, P1). Also, according to a recent Gartner survey, 71% of enterprises convert the legacy developers to be object-oriented software developers. There was a survey (Feiman P7):

Migrated legacy developers to Java	71%
Hired new Java developers	69%
Partially staffed teams with developers from external service providers	40%
Completely outsourced projects	22%

1.Training process
To become a proficient object-oriented software developer, developers must take at least five courses in object-oriented software concepts and object-oriented languages.

Course	Duration (days)	Curriculum
OO Paradigm	2	Encapsulation, polymorphism, class hierarchies frameworks
OOA&D	3 to 4	OO modeling methodology, e.g., UML
OOA&D tool	1	Diagramming, scripting, code generation, reverse engineering.
Java IDE	5	Visual programming, team development, configuration Management.
Java Language	5	Basic concepts and techniques, applet programming.
Java to DBMS	3	Object persistence in relational DBMS, class to tables mapping, ODBC, JDBC.
OO Project Management	1 to 2	Iterative and incremental process vs. waterfall process.

The entire training program is necessary. If this training is not provided, “hidden training will take its place. The programmer or project manager will be self-studying the same disciplines. Also, making mistakes will not be avoided. The total “hidden” training time will be much longer than if all formal training is completed.

D. University Education

The second indirect view of complexity is based on introductory programming courses at Grand Valley State University. The topics from three courses are compared below. The Pascal and Java courses are standard versions of the Computer Science I course mandated by the Association for Computing Machinery (ACM) curriculum guidelines. The Visual Basic course closely parallels these, and is a service course for other departments.

CS 151 (Pascal)	CS 160 (Visual Basic)	CS 162 (Java)
The Pascal programming environment. Editor and compiler.	The VB programming environment: various windows and tool bars. VB Controls.	BlueJ programming environment.
	Graphical User Interfaces. Event-driven applications.	
Input/output with READ, READLN, WRITE, WRITELN	Input/output with text boxes and labels.	input/output with System.out.println, and BufferedReader
Assignment statements. Arithmetic operators.	Assignment statements. Arithmetic operators	Assignment statements. Arithmetic operators
Scope Variables, pre-defined data types.	Scope Variables, pre-defined data types.	Variables, data types
Sequential program execution. Flow charts.	Sequential program execution. Flow charts.	Sequential program execution.
Conditions, relational operators and logical connectives.	Conditions, relational operators and logical connectives.	Conditions, relational operators and logical connectives.
Selection: If-Then; If- Then-Else; nested If statements, Case.	Selection: If-Then; If-Then-Else; nested If statements, Case. Check Boxes, Option Buttons, and Frames.	Selection: if; if,Else; nested if statements, switch
Repetition: computed, pre- and post test loops.	Repetition: computed, pre- and post test loops. Scroll Bars, List Boxes	pre(while), post(do) and for loops
Procedures. Formal and actual parameters.	Procedures. Formal and actual parameters.	Procedures. Formal and actual parameters.
Formatted I/O (Format statements)	Output with Picture Boxes. GUI considerations.	formatted I/O (NumberFormat or DecimalFormat)
Arrays of variables.	Arrays of variables, controls.	Arrays and ArrayLists
User-defined data types, Records, and Fields.	User-defined data types, Records, and Fields.	Classes and objects. Encapsulation.
File Input/Output. Sequential and Random Access files.	File Input/Output. Sequential and Random Access files.	Sequential file I/O
Searching arrays (linear and binary).	Searching arrays (linear and binary).	Searching arrays
Sorting arrays.	Sorting arrays.	
		Inheritance, has-a vs. is-a relationship
		variable scope, static variables, call-by-ref vs. call-by-value

The Pascal and Visual Basic courses represent three semester hours of course credits (28 lecture hours and 14 lab hours). The Java course carries four semester hours of course credit (42 lecture hours and 14 lab hours).

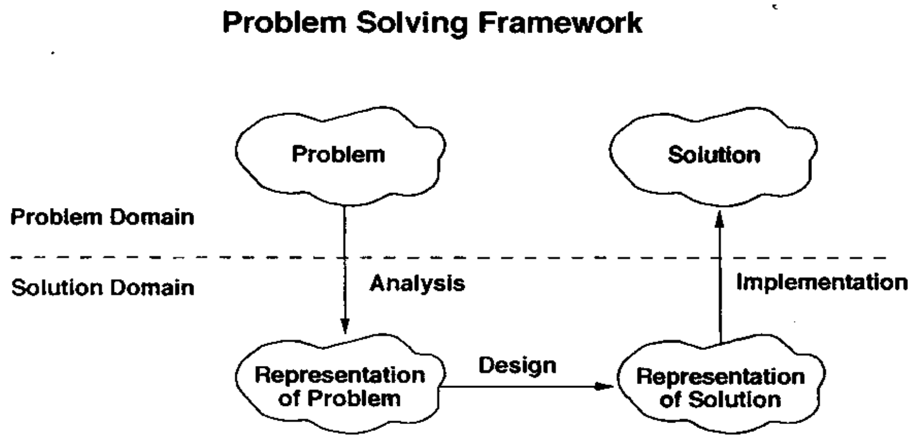
E. Conclusion

Whether measured directly or indirectly object-oriented software is more complex than functionally equivalent procedural software. The promise that more natural description and encapsulation will result in less complex software has not been realized.

V. The Promise of Naturalness in Respect to Object-Oriented Programming

A. Natural Conceptuality and Abstraction.

Naturalness, in this context, means thinking in terms of things that naturally occur in a subject, a workplace or any field of endeavor. Of course, for this paper the focus of the discussion is on areas where it may be expected to want to find a solution to a problem or desire, by writing a software program or system. In a broader sense, this restriction is not strictly necessary, but it will not serve the to enhance clarity of our purpose to include the much wider scope in which Object-Oriented (OO) concepts may be found to be useful.
That purpose being to examine the success in the use of OO processes and ideas in grasping the abstractions and concepts of the problem sets or domains to which it has been put to find satisfactory solutions.


Problem Solving Framework
Problem
Solution
Problem Domain
Solution Domain
Analysis
Implementation
Representation of Problem
Design
Representation of Solution

B. Abstraction and Objects

One of the promises of Object Oriented development was that the developers can work, and the solution set could be framed, in the context of the problem domain. In other words, rather than the software solution having to end up looking like and conforming to, the architecture of the machine it is to run on, it can be developed in the language, terminology and frame of mind of the field or discipline, for which it's designed.
Objects, if chosen properly, directly represent the parts of the problem for which a solution is trying to be built. The relationships between the objects, model the structure of

the problem domain in which the user is working, and the object's methods represent the behavior of the objects in this structure.

Abstracting the structure and behaviors of the problem environment or domain in this way makes it more "naturally" comprehensible to the user or domain expert. Using abstractions like this for describing the problem to be solved has the effect of speaking in terms that are familiar to the user. It also helps form a bridge between user and developer with a common problem domain terminology and language.

This allows the user to participate directly in the discussion of the problem and what solutions may be effective and acceptable. It brings the user onto the team as an active participant rather than just using him or her as a data source and promotes more and better communication between users and developers.

> "It [an Object] allows you to package data and functionality together by concept, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine." (Eckel 2000, p. 38)
>
> Casting the solution in the same terms as the problem is tremendously beneficial because you don't need a lot of intermediate models to get from a description of the problem to a description of the solution. (Eckel 2000, p. 40)
>
> An OO expert, Grady Booch, put it this way, "An object-oriented model of the problem and its solution encourages the creation of a common vocabulary between the end users of the system and its developers, thus creating a shared understanding of the problem being solved."(Booch, 1996, p. 24)

Since it is generally accepted that a project or system's requirements are subject to change as the project develops and that they are not static entities, to be decided on at the beginning of a project and changed 'never thereafter', it can assumed that changes and modifications throughout the course of development will be a normal and expected occurrence.

It seems obvious that it would be to the advantage of all parties involved that the methods and substance of communications be as simple, clear, straightforward, and unambiguous as they can be made. Further, as pointed out above, it's not just the initial requirements that benefit from clear and unambiguous understanding of the system and its components and processes. Accurate and clear communication of ideas is vital to the successful development of a software system throughout the entire development effort.

This is true, of course, whether an OO approach is being used or not. But it is part of the basic premise of this paper that Object Oriented development proponents have claimed that the methods, tools and mindset used in OO give rise to better understanding of a problem domain, quicker results and easier paths to solutions.

One of the questions being addressing here is whether or not that is true. And if so are practitioners actually able to accomplish these results in the real world or does it remain for some reason, an unfulfilled promise?

But if it's supposed to be easier, quicker, better, propriety requires at least some attention to the question "better than what?" or "better how?" How is a procedural development approach (SA/SD) different?

C. The traditional alternative- the Procedural paradigm

Structural Analysis/Structured Design is predicated on top-down algorithmic decomposition. This, by definition, is focused on the algorithms, -- based on functionality or procedure (method in OO). The developer focuses on decomposing the system or system model in terms of the algorithms of which it's composed, into finer and finer granularity until an atomic level is reached. At this level, no further decomposition is necessary or desirable. The system or subsystem is divided into units that perform a single task or are fully self-contained.

Once the analysis is complete, the design of the system can be addressed, creating the necessary algorithms and combining them in functions. Functions call other functions and progress is made by creating, one by one, the functions that will achieve the desired result. Beginning with step 1, then on to step 2, then step 3, and so on until the finished result is reached (stepwise refinement). In theory, it works well. Everything is well organized and under control. But it has been noted that it doesn't always work that way in practice. Some project situations just don't lend themselves to this kind of prescriptive planning.

> "Structured programming can seldom design what the completed system would be without actually implementing the system. If design were found to be incorrect after programming has started, then the design would have to be entirely restructured. And that would cost corporations time and money. The difference between structured programming and OO programming is how the data and functions are kept. In structured programming, data and functions are kept separately. Usually all of the data are placed before any of the functions are written. Sometimes, it is not intuitively known which data works with which function. But in OO programming, the related data and functions of an object are placed together within one unit (Montlick p. 1).
>
> With the object-oriented approach, the design of the whole system could be modeled at a higher level. Any potential problems with the design can be fixed at this level without having to start any programming. Also, people can easily understand the system as objects rather than procedures since people think in objects. For example, people see a car as a system with an engine, gas tank, wheels, etc. But most people would not see a car as a series of procedures that makes it run. Since it's more natural to think of a system in objects, it is understandable why OO technology is gaining popularity. "(Lam 1997)

The attempts to find system solutions, where requirements are uncertain or subject to change, has lead to the rediscovery and recent rise in popularity of OO development, which by all accounts, handles this kind of problem rather well.

Still the published and unpublished opinions from proponents of procedural systems development seem, in a number of instances, to exhibit a feeling of being threatened by OO ideologies. There appears to be a definite thread of reacting in anger against the enthusiasm (sometimes over enthusiasm) of OO proponents. Some authors go so far as to engage in what appears to be willful misunderstanding and misapplication of OO tenets and procedures in order to show how foolish and misguided the OO approach is, in light of 'the holy tradition of procedural development.'

> "Real world objects do not have a list of **predetermined** "proper" behaviors associated with them, which is essentially what OOP does.
>
> OOP was born in the domain of physical modeling, where one strives to make nouns "self-handling" more or less.

> If the real world was like OOP, then we would get scenarios like, 'I am sorry Mr. Mugger, but I have no victim.mug() method. You will just have to try somebody else.'
>
> x = a + b
>
> Even a simple math equation exposes the artificial or forced association of behavior to one-and-only-one noun (one class per method). The plus operation (+) must "belong" to either *a* or *b* in most OOP. I don't know about you, but I view "plus" as a relatively free-floating operator that takes two operands." (Findy Services & Jacobs, 2002)
>
> and
>
> "[…] OOP does **not better** model the real world than other paradigms. If anything, it is a worse model of the real world in this regard. The real world does not have compile-time checking for the most part." (Findy Services & Jacobs, 2002)

As entertaining as they may be, opinions and objections of this nature should be ignored due to the irrational nature of the reaction of the author in favor of more professional research and study.

D. Focus on Communication

So there is the claim that modeling the problem in terms of the problem-space –in Objects and their methods or behaviors – promotes better communication between stakeholders. Also, that it is normal and natural for requirements to change during the creation of a solution for the initial problem being dealt with –development. How does OO achieve its claim that the "natural" nature of its abstraction techniques is superior to those of traditional structured design?

In an Object-Oriented environment the solution will be defined and developed in terms of the problem domain. In other words, using for example, accounts, orders, transactions, balance inquiries, order line items, etc. rather than list pointers, array elements, hash registers, conditionals, looping structures and database queries, and other machine oriented structures.

But no matter how rational or intuitive it may seem, it can't just state that objects and their methods, combining to form classes are the answer to the question, and leave it at that. That just because it sounds reasonable, that it must be the answer. There must be some way to realize this conceptuality. There must be some way that the abstractions can be seen and manipulated by the stakeholders to be a focal point for communication and discovery. Perhaps a notation would serve here. It would have to be rich and flexible enough to easily convey a widely varied scope of information and simple enough to be easily understood without extensive training.

Of late there has been some work, or at least investigation of the idea of a universal grammar. That is the idea that there may be some fundamental or even genetic source of the rudiments of communication and language. Recently a brief study was done to test the comprehensibility of a series of diagrammed notational phrases. Each of the study participants was to view each of the phrases and select what he thought was the meaning of each phrase. The results of the study were, that in a great majority of cases, the frame of mind of the participant had an obvious influence on what they thought the phrase meant. Obviously there was a strong tendency to read into the diagram or model what the reader wanted or expected to see, rather than what the phrase was intended to mean. This shows that the paradigm of the reader has a greater effect on meaning, or apparent meaning, than the intent of the writer. (Ralph Palmer 2002)

Research in this area could help promote improvements in communication between software development stakeholders, the importance of which we have and will continue to discuss. Another perspective on this is whether work on, and with, the use of software requirements and design modeling tools such as UML might further the discovery and development of a universal grammar.

1. Accuracy of Communication – Understanding each other

It has been stated that the vocabulary of a domain can be found in the abstractions of that domain. (Booch 1996) So, to discover and use that vocabulary it is necessary to find a method, sufficiently clear and precise, to represent the abstractions that define the structure of the problem domain.

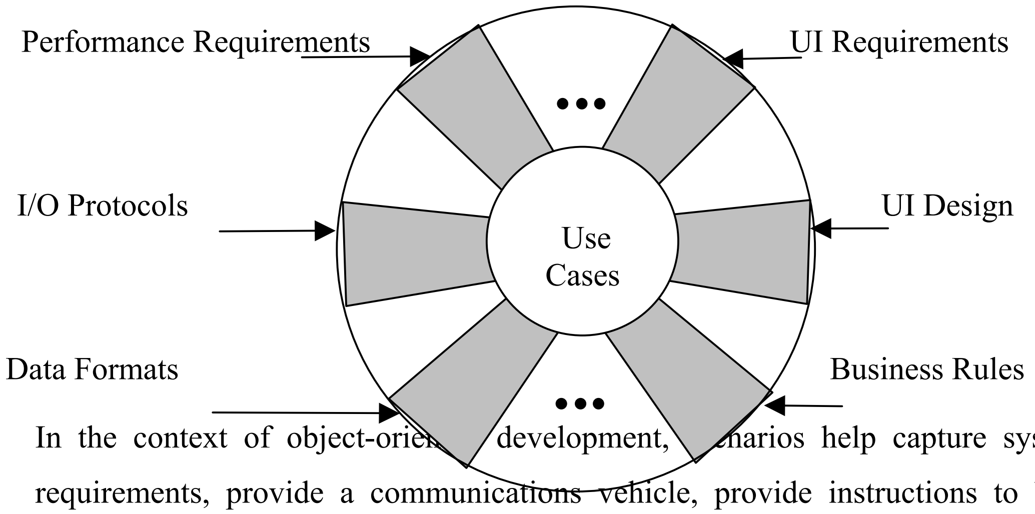
Object Oriented development claims to be able to do so in such a way as to use the terms and mind-set of the problem domain or user, rather than those of the

architecture of the machine on which the software solution will run. Although it seems this naturally leads to an easier understanding of the structure and behaviors of the system being constructed, in order to realize this naturally convenient conceptual abstractive ability, we need to show the existence and use of some tool or vector to carry this idea objectively.
In order to show that OO has achieved even a measure of success in improving the clarity of defining the problem and solution sets, improving communication, shortening development time, or making programming solutions more robust, and easier to maintain and enhance, it is necessary to show how this has been applied in a practical sense. Through what means can or has this been realized?

2. A Good Tool – to the rescue

The tool of choice seems to be Ivar Jacobson's use case. It is not an OO specific tool and has been around for quite a while. A use case is comprised of one or more scenarios. A scenario is a process thread. It is a task followed through the problem domain from start to finish. (Eckel 2000, p. 77) explains it as what results from a question like "What would a teller (or other primary object or role player in the problem domain) do if …" It defines and expands, or "exposes" a behavior in a system, so that it can be suitably discussed between developers and users.
Use cases also help structure project planning and design information such as teams, release dates, priorities, user interface and system tests, development status, business rules, protocols, formats and other such items.

All these requirements are directly connected to, if not contained in use cases. The analogy given by Alistair Cockburn, a current expert in the field, is that of a wheel, with use cases as the hub, connecting all the other information, depicted as spokes of the wheel, all heading in different directions. (Cockburn 2002, p. 15)



In the context of object-oriented development, scenarios help capture system requirements, provide a communications vehicle, provide instructions to both developer and testing teams and “form an essential basis to the scheduling of iterations during design and evolution”[implementation]. (Booch 1996)

Booch continues to explain that having “users and domain experts focus on scenarios on the level of the problem domain” “helps them avoid the temptation of diving into solution-oriented abstractions.” And by having developers focus on scenarios, it forces them to get a basic grounding in the vocabulary or language of the problem domain and “consider an intelligent distribution of responsibilities throughout the system.”

E. Realizations

After studying and using use-cases and scenarios one would have to agree that they are very useful tools for gathering requirements. When done properly they provide an extremely clear and unambiguous description of a proposed system’s behaviors. Having that, a developer can design a system who’s structure is efficient in creating a software solution and code classes that implement just that behavior, a tester can test whether or not those behaviors are correctly present in the developed code.

So, not only is a clear and accurate requirements specification achievable from their use, but also a design structure, an implementation framework and a plan to test against can all be derived. And when this work is finished, the collection of use cases form an invaluable part of our system documentation.

Now that it has been discovered that there are indeed tools that allow the use OO in a manner that will allow the fulfillment of its promises of greater ease of conceptuality, abstraction, communication, what evidence is there that it has actually been realized in practice?

> "The application of [Object-oriented systems development] OOSD is usually expected to result in many distinct advantages over [conventional systems development] CSD.
> However, there are also reports of problems associated with the use of OOSD. Evidence of the advantages and disadvantages comes primarily from three sources:
> (1) the expert opinion of practitioners and consultants (Booch ,1994; Coad and Yourdon, 1991; Coleman et al., 1994; Jacobson et al., 1995; Rumbaugh et al., 1991),
> (2) case studies (Berg et al., 1995; Burkle et al., 1992; Fayad, et al., 1994; Filman et al., 1992; Rumbaugh et al., 1991; Taylor 1992), and
> (3) empirical research (Basili et al., 1996; Boehm-Davis and Ross, 1992; Chen and Chen 1994; Davies et al., 1995; Hardgrave and Dalai ,1995; Herbsleb et al., 1995; Lewis et al., 1992; Pennington et al., 1995; Vessey and Conger, 1994; Wang, 1996).
> Some of the commonly reported advantages of using OOSD are:
>
> • An easier modeling process
> • Better analysis and design models
> • Easier transition from analysis to design to implementation
> • Improved communication between developers and users
> • Improved communication among developers
> • Easier, more flexible development using software components
> • Decreased development time
> • Higher system quality
> • More effective forms of modularity
> • Increased reuse of analysis and design models
> • Increased reuse of program code
> • More stable system designs
> • Less system maintenance
> • Easier system maintenance
>
> The most commonly cited disadvantages of using
> OOSD are:
> • Increased development time
> • Poorer run-time performance" (Johnson, Hardgrave & Dok, 1999)

There are conflicting reports concerning some of the advantages, as well as efforts to improve the disadvantages listed in the quote above. In particular, the apparent contradictions seem to be in the areas of system maintenance and design and code reuse. Notice too that development time is listed in both advantages and disadvantages. It would appear, if taken literally that the use of OO development methodologies both increases and decreases development time. How can that be?

It may be that over several studies and a number of instances, conflicting results were reported. It may be a matter of proper implementation of the practice. It may be that there are times and situations when it is appropriate to use an OO methodology, and times and situations where it's not. This may be one of the things that remain to be worked out within the realm of improving software development practices and processes in general. Research results such as this confirm that there's still a lot of room for improvement in the field.

This part of the paper is predominantly concerned with the first five items in the "industry analysis" quote above. There seems to be sufficient evidence that theses claims have merit. It can also be noted that in the case of each item, the natural frame of thought can be argued to contribute significantly to the successful attainment of the stated advantage.

> "Object Technology International (OTI) has been doing business since the late 1980s contracting to deliver hard real-time systems in Smalltalk, "on time or you pay nothing." The company's president, Dave Thomas, relates that at the time they started, no one would contract with them unless their project was dangerously late and a company was desperate.
>
> Using experienced developers and a custom development environment, OTI met every project deadline and performance requirement over a 10-year period.
>
> They were able to demonstrate that Smalltalk could be used successfully for hard real-time systems with tight performance constraints, including waveform generation on an oscilloscope." (Cockburn 1998, p. 17)

Mr. Thomas is a long time adherent to OO technologies and currently a member of the Agile Alliance, a group of software development experts and methodologists dedicated to the promotion of light weight OO development practices and methodologies. The history of his success in the use of OO procedures and practices, as noted above, points up the fact that the use of the “natural” paradigm involved in OO development can be successful and effective in software development. In fact, the team cited in the quote above is involved in fine-tuning it’s OO procedures. The inherent naturalness used in their methodologies is a second nature given in their case. Something taken for granted as the best way to approach creating solutions for their customers.

F. Conclusions

It has been shown that there is, indeed, evidence to the effect that OO procedures have been used successfully in developing software systems, and that the ease of abstraction and conceptualization available with OO methodologies can be used to good effect.

This is especially true when use cases are part of the project toolset. The natural ease of using this tool in analyzing and defining the problem at hand, framing a design, organizing solution implementation and creating and implementing test plans is obvious, even to someone untrained in the field, a fact which further proves the point.

The developer needs to understand the problem to be solved. The user has the information the developer needs. As stated earlier, a common vocabulary needs to be found between the user and developer. Some way must be found so as to allow the abstractions to be seen and manipulated by the stakeholders, to be a focal point for communication and discovery. Use cases provide that focal point. They do so in a way that is natural to, and understood by the user, the domain expert, and which allows the developer to expose or discover the underlying abstractions of the problem domain.

Those abstractions are then used to create an object framework. That framework will be used as the basis for the problem solution. This keeps both the problem and solution domains in a form that the user may more easily comprehend. One in which the user is familiar with and so is able to better understand. This means that the user is much more likely to receive a finished system that more accurately addresses the original problem. It also means that the developers are less likely to encounter false paths and dead ends in their efforts to arrive at a workable solution to that problem.

In other words the “naturalness” of OO concepts used in conceiving of and working with problems and solutions can be a valuable asset if applied properly and in the proper situational context.

It must be recognized that no solution, how ever good, can be successfully applied in ALL situations. But, the evidence is there to show that the OO approach to abstractions and conceptualizations of problems and solutions, does work and has provided an alternative and in many cases, a faster, easier and more accurate means of handling software abstractions and conceptualization.

VI. Powerful Modeling

A. Introduction

In the late 1980's, authors such as Grady Booch, believed that a software crisis was looming. The cost of hardware was decreasing while software demands increased. A 1982 article by Datamation shows the costs of software changing from less than 20% of system cost in the 1960's to over 80% in the 1980's. In addition to costs, many large software projects fell behind and/or delivered only part of the originally required system (Booch 1987). Cox and Novobilski cite a study by the US Government Accounting Office (GAO) from 1979 showing that less than 2% of studied systems were actually used as delivered (Cox 1991). At the same time the complexity of systems continued to grow unabated. As hardware increased in speed and capacity, software grew in both complexity and size. Studies have shown that the complexity of a 100,000 line system is much more than just 10 smaller 10,000 line systems (Booch91). Programmers had turned to object oriented programming to solve the coding crisis.

However, as the demand for systems grew, and as programmers used OO programming to meet that challenge, the demand for a way to organize the requirements, the language and the tools also grew. Software Engineering, which had existed well before the advent of OO programming, was required to meet the combined challenge of larger systems, new programming styles and high user demands. Patrick Loy, in his 1989 comparison of OO methods and Structured Development (SD) methods notes that "it is an undeniable truism that the methods and the tools used during the software development can have a significant impact on the quality of the final product." He also notes that OO is being heralded by some as the silver-bullet for software development, but that it is creating a lot of interest in Object-Oriented Development (Loy 1989).

B. Promise

The waterfall and other structured methods were developed to meet the early challenges of software engineering. These methods typically used a rigid set of stages or phases to complete the project. While the number of phases varied according to the method or the practitioner, they usually began with some sort of requirements documents, which were then signed by the client before progressing. The second major milestone (encompassing one or more stages) was analysis. For the most part this was decomposition of the requirements and a further understanding of what the project entailed. This too was negotiated and signed off on before the next step. The third major grouping was design, where the analysis, which was intended to have been solution neutral, was used to set the blueprint for the solution. After design came a coding phase, then a testing phase, sometimes included in the coding or development phase, sometimes having multiple phases of unit testing, integration testing, system testing, and user-acceptance testing. After each phase or milestone, typically the client and developer community would meet and verify the effort so far. While this methodology improved the final product, the result was that changes to requirements were difficult to accomplish and the initial requirements document really relied on perfect foresight.

While the roots of OO date as far back as the 1960's the history of analysis and design built for OO did not really begin until the late 1970's or the early 1980's. In 1980, Grady Booch realized that there was a need for software engineering to specifically support the language syntax and concepts into the Ada language. Booch, utilizing the work of Russell Abbott, approached design using nouns and verbs. Booch formalized Abbott's work and referred to it as "OO design." By 1986, Booch revised his thinking to include more than just design and he called it a whole process for development (Berard 1995). Booch was convinced that OOD would lead to software of higher quality and that was more likely to meet the requirements for which it was built (Strong 1992).

Constantine noted in 1989 that the major difference between Structured Design (SD) and OOD was role of data and functions. In SD, the functions and the procedures were considered first and the data later. Furthermore, functions and data are thought of either independently or attached to the functions. On the other side of the issue, argues Constantine, is the OO paradigm. Here data is the primary consideration and functions are attached to their related data, problems were looked at in a new light, that of how the data related in terms of classes, and the rules that they obey. (Loy 1989).

OOD literature of the time exhibited a great deal of enthusiasm and hope. It was described as revolutionary instead of evolutionary and Booch claimed that it would lead to improved maintainability and understandability. There were typically two reasons for this optimism. First was that the thinking process that OO was built upon was more "natural" understanding. Secondly, it was believed that the modeling of the problem space more directly mapped to the solution space in OOD than it does in SD. (Loy 1989). In 1991, Brian Berry noted that everyone from the analysts to the programmers was being encouraged to use OOD modeling because the "OO language allows the models to be directly released into code, so no intellectual gymnastics are required to move the worlds of the analysts, designers and programmers (Berry 1991).

By this time, other ideas and approaches had been published and debated. However in general, OO methodologies can be thought of as having come from the OO programming languages and the design modeling (Dawson 1998).

In 1994, Booch and Rumbaugh started collaborating by forming the Rational Inc. company and worked to unify their models. Later joined by Jacobson, the end product is what is now called Unified Modeling Language (UML). Some now believe that UML defines the industry standard for a formal notation and semantics for use in an OO technology. It is so prevalent that most texts and tools include UML. Its strengths are that it provides a "common and consistent notation" used to describe systems (Ambler 2002). However, it must be noted that UML is and remains a package of tools allowing many different diagrams.

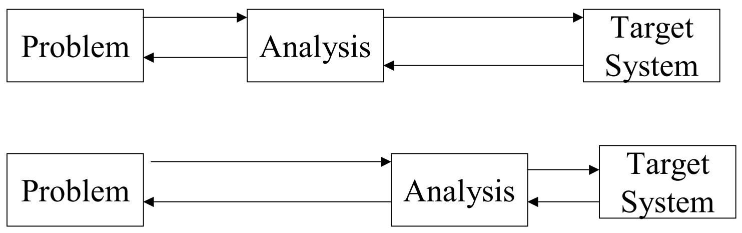
In 1993, it was argued that OO Analysis (OOA) was still lacking. New methodologies and models such as the fountain model had been developed especially for OO development. It appeared that while the boundary which distinguished analysis and design had been blurred, and in some models the stages interleaved, each process still existed.

Analysis is the description of what the problem is and what the user requirements exist. Design is based on the construction of the solution that will meet the requirements that were noted in the analysis phase. Delving further into the analysis model, it should encompass four interdependent aspects:

1. The expectations from the ultimate user regarding the entire system and how the systems and the outside interact.
2. A data dictionary to define the terms, meanings and properties of the application.
3. What the environment requirements are or what the external interactions will be.
4. The requirements of the operations computer system should perform.

Furthermore, analysis should consider aspects of Quality Assurance. As errors in the finished system are much more costly to correct than errors in analysis, OOA should include both verification and validation. Verification can be defined as checking to ensure that the requirements will produce a consistently correct result inline with the user demands. Validation is the activity where the requirements are checked to ensure that is what there is a direct correspondence between the user's statements of desires etc. and the model shows (Hoydalsvik 1993).

One of OO strengths is said to be that it is more natural, or closer to the problem domain. That strength is based mostly on the difference between problem-orientation versus target-orientation.



Problem orientation (top) Target-system orientation (bottom)

Target-orientation tends to fail to find potential needs for changes in the external side of the system, which are required. This leads to gaps and user dissatisfaction. It also tends to imply design decisions before the problem is fully understood. Finally, maintenance becomes more difficult because the analysis models have emphasized the description of the system, not the interaction between the users and system. Target analysis puts the bulk of the problem onto the analyst to convey all the meaning of the user to a target environment. The user is also taxed due to the challenge of understanding whatever the model shows and to make sure that is truly desired. Therefore the user has a hard time with QA (Hoydalsvik 1993).

The benefit of target-system analysis is that it makes the transition to design much easier, however, this does little good if that analysis is not understood or is incorrect due to challenges of the user (Hoydalsvik 1993).

OOA claims to meet that challenge by moving the burden from the user, by making the analysis problem-orientated. This means the models produced are closer to the problem domain and the users would more naturally understand the requirements and the structure of the system. This should allow the user to better perform the QA requirements while understanding the problem (Hoydalsvik 1993).

Finally, given the current rationale for using OO technology, namely that OO provides a faster, better, cheaper and more robust system, it is believed that these are enhanced if they are understood early in the process (before design and coding). Simply using OO technology and coding will only allow for partial benefit. To gain the best benefit, an entire software engineering process must be used and include OO specific analysis, design etc. (Berard 1990).

Using an OO approach to software engineering will improve tracibility (i.e. tracking the requirements through the phases) and reduce integration problems while improving the integrity of the process and the product (Berard 1990). Additionally, claims about the OO paradigm include ease of understanding, higher reuse due to having previously designed systems. Dawson also notes that it is generally accepted that a strength of the OO methodology is that complexity is reduced due to tracibility and the modeling of the interaction between objects (Dawson 1990).

The early hopes of the system appear to be best summed up by Brian Henderson-Sellers and Julian M. Edwards who conclude that using the same environment that features OO design from requirements analysis to implementation should bind together the various phases of the project. In addition, they claim that “the result should be a more maintainable system, a more extensible (flexible) system closer to the user’s requirements and a system that requires overall less total time to produce and maintain (Henderson-Sellers 1990).”

C. Actuality

Since 1988, multiple methods (at least 19 by one count) have been proposed to replace the SD of the previous era. OOA, according to Peter Coad, uses modeling to establish and show the "problem domain classes..." OOA will show what each object in a class is and what it does and what it needs to know. It additionally uses the same model to specify interactions between objects with the OOA resulting model feed into the OOD. In design, additional details on what the class needs to know and what it does are added to the same model. (Coad, 1993)

Another challenge is the claim of Hoydalsvik (1993) that the OOA process is more problem-oriented, i.e., closer to the user. Specific case studies indicate that analysts believe that the models such as Object Modeling Technique (OMT) or UML cannot be sufficiently understood by clients. (Dawson 1998)

Some of the hopes for improvement by using OO methodologies and OO language may be linked to social pressure instead of its advantages. System developers may experience pressure from both inside and outside the IT organization to use OO system development methods. Authors, consultants, vendors, and potential future employers may have a "profound impact on the acceptance of OOSD…." And managers, swayed by hype and marketing, can have a strong impact on the decision to use OOSD. (Johnson 1999)

Even when used, the advantages of using OOSD have seemed to change. Contrasting the expert opinions of the past stating that OO methodologies will improve the process, Johnson (1999) notes that commonly listed disadvantages of OOSD include increased development time. Part of the problem may be that some methods are considered underdeveloped in the setting up the specifications of the external functions as they relate to the system of the whole. This appears to be the downside of the well-developed object functions. The same thought extends that OO methods are insufficient to break the system into subsystems. (Wierienga 1998)

UML is, according to at least one author, useful, but not sufficient for development of software for business. Scott W. Ambler of Agile Modeling argues that UML lacks a user interface diagram. This interface will explore how users will interact with the system from a high level and allow or point out important questions regarding an interface. On the other end of the process, UML does not yet include a data model (although one may be added.)

However, the larger shortcoming is the lack of a viable executable model. The executable UML is a vision from its beginning that would allow a smooth transfer from analysis and modeling without the translation problem of moving from analysis to design and code, steps that often introduce deviations from original requests. It would have also allowed changes to the system as requirements evolve and delay implementation. Problems that may exclude executable UML include the previously noted lack of models; without those models anything generated will be incomplete. Secondly, the wide range of available platforms and languages will make a collection of tools generated from a large range of vendors that will most probably include proprietary extensions and integration difficult if not impossible. On the other end, if a single vendor were to attempt a tool, it would most likely be too narrow. But without executable models, we still have translation problems from phase to phase. (Ambler 2002).

There is also evidence that analysts believe that clients and users both find models (such as UML or OMT) to be “much too complex” from both a technical and conceptual perspective. While the formal models were difficult to use, the informal models that were created in the process of making a formal model were generally more understood by the user and were actually used to both the validate of the system and to pass on to the design stage. This however, will reduce or minimize both the tracibility of the model from phase to phase and the gains from using formal methodologies. There is some evidence that analysts must continue to use informal methods and natural language to communicate with users but continue to develop formal models to pass on to further stages. However errors that arise due to missed specifications, inconsistencies and misunderstanding often result in costly post-implementation fixes or even failure of the system. (Dawson 1998).

While the formal notations aspects of UML and OMT are difficult to be understood by the user, informal diagrams, interaction diagrams, and Use Case variations are better understood and are a more appropriate method to convey requirements and analysis with users. Ivar Jacobson, the author of use cases, is reported as a “first system model” because “ […] object models are too complex” for people outside the development community. However studies have seemed to indicate that while Use Cases have value they are used in a less formal setting than Jacobson suggests and often usage is outside the standard methodologies. (Dawson 1998).

Regardless of the model used, whether it is UML or other, methodology including modeling should not be treated lightly. Managerial pressure to control time and money combined with the natural tendency of developer to rush to code. In addition to the expense of time and money, the process of requirements gathering and analysis requires a lot of skill both in the problem domain and the OO domain. The result is that except for a few cases of constrained domain or trivial applications, the OO process of defining requirements is “almost guaranteed to result in incomplete specification [and] incorrect specifications.” A study found that requirements documents are typically only 7% complete and 15% accurate. However, as time and money are dedicated to the art of analysis, an old problem develops as requirements change over time and the cost of changing rises. (Atomic Objects 2001)

D. Conclusion

With all the changes in programming and modeling in the past years, one axiom still holds true. The models and methods employed during the entire system development process have an overwhelming effect on the final product. The requirements and analysis phases remain a vital part of the process. In fact, having a good process and following it appears to be better than using any new super process. Far too often a rush to begin coding interferes with the process, crippling it even before any OO classes are coded.

Some positive effects of the OO modeling exist. The OO concept of abstraction helps to create better models sooner. Yet, modeling, for all its positive additions to the requirements and analysis, is still tied to the human use of it—the implementation.. Ralph Palmer demonstrates that people will try to read a model the way they want as opposed to what it really says. That is, should a model clearly show that GIFTS receives PERSON; the natural tendency to read it as a PERSON receiving GIFT will win out the majority of the time. (Palmer 2002)

It appears that the promise of powerful modeling has been partially met. Many new tools (such as UML) have appeared. Their influence has allowed the analysis and requirements phases to be completed in a uniform matter. Other methods such as Use Cases have additionally improved the tools available. These additional tools have allowed the possibilities of OO to shine. Used properly, they enhance the process and minimize the errors. Unfortunately, it appears that Use Cases are not always used to its best advantage.

Still, in another light, it appears that OO has yet not been the silver bullet that was once hoped. The modeling techniques as a whole were developed after the language and while that itself is not bad, the effect is that modeling continues to evolve and develop. A combination of poor management, a rush to code and inexperienced modelers combine for recipe for a failure – or at least the continued creation of sub-par systems. Still other promises, such as executable models have not yet hit mainstream and continue to be hope for the future.

VII. Object Oriented Software Development and Extendibility

A. Introduction

Software specifications are as accurate and fixed as they can be in the beginning stages of the development cycle. Based on requirements, these specifications are written down, in contract form, with the hopes that the customer first understands them to be correct from their point of view and ultimately accepts them as what will be the expected functionality of the software when complete. Unfortunately, developers and customers are human and are subject to interpretation differences. When customers read these specifications, they have to interpret them to accept them. This interpretation and agreement process will at times lead to acceptance in the beginning, but what is delivered is not what was expected. What does this mean to the development process? Development times and ultimately costs are increased due to changes in specification. Designs are changed, implementations are reworked, and the whole testing process needs to be repeated (Sommerville 1992). Does Object Oriented (OO) technology reduce the cost of late software changes due to specification changes? In particular, can OO extendibility and its promise for "ease of adapting software components to a change in specification" result in reduced cost (Nierstrasz, 2002)? This paper will outline common types of extendibility one might deploy. It will use for demonstration purposes a fictitious OO project to develop a student management system for an intermediate school district (K-12), highlighting the pros and cons for each application of a particular extendibility type when faced with having to adapt to a change in specification.

B. Extendibility Defined

Extendibility in software development manifests in the language used. In OO languages, extendibility is realized through inheritance and subclasses. Before this paper progresses, these attributes should be quickly defined. "In programming languages, inheritance means that the behavior and data associated with child classes are always an extension (that is, a larger set) of the properties associated with parent classes" (Budd 1997). Subclasses are the child classes described previously. They are classes that inherit behavior and data from parent classes. There is a rule called "the principle of substitutability" that helps one further define a particular category of subclass. This rule reads as follows: "if we have two classes, A and B, such that class B is a subclass of class A (perhaps several times removed), it should be possible to substitute instances of class B for instances of class A in *any situation* with *no observable effect.* The term *subtype* often refers to a subclass relationship in which the principle of substitutability is maintained" (Budd 1997). This type of subclassing is desirable for ease of testing and validation and therefore should be considered when attempting any of the following extendibility applications just for the mere time and cost savings it provides, not to mention readability and understandability.

C. Common Inheritance Models

The common forms of inheritance this paper will explore (in order) are listed below:

Specialization	The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
Specification	The parent class defines behavior that is implemented in the child class but not in the parent class.
Construction	The child makes use of the behavior provided by the parent class, but is not a subtype of the parent class.
Generalization	The child class modifies or overrides some of the methods of the parent class.
Extension	The child class adds new functionality to the parent class, but does not change any inherited behavior.
Limitation	The child class restricts the use of some of the behavior inherited from the parent class.
Variance	The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.
Combination	The child class inherits features from more then on parent class. This is multiple inheritance...

- (Budd 1997)

D. Subclassing for Specialization

The first application of inheritance to explore is when the child class is a specialized case of the base or parent class (Stroustrup 1991). In this type of subclassing, the child class simply inherits without change the base classes behavior and data and adds its own behavior and data to fulfill its specific needs or requirements. An example might be where one starts with a base class Student. Student encapsulates firstName, lastName, gradeLevel, courseSchedule and courseGrades for data and some simple getter and setters as well as computeGPA() for behavior. The superintendent of the school district believes through specification that all types of students will be handled, but when presented with the working software components, he or she realizes that students with special medical care needs were not addressed. In this situation, the development up to this point could be extended by specialization and the new subclass SpecialNeedStudent could be introduced. In addition to behavior and state inherited by Student, SpecialNeedStudent might have additional data such as medicineSchedule and behavior such as getNextDosageTime(). Because this type of inheritance follows the principle of substitutability, it can be concluded that the use of Student or the use of SpecialNeedStudent up to this point will go unnoticed in the current software components. Thus, the tester can gain immediate confidence in the ability to validate this change across existing execution paths through the subtype principle. A downside to this type of subclassing is known as the "reverse polymorphism problem" (Budd 1997). In many OO languages there are collection classes such as ArrayList in Java, into which one can place like types (Student). If the developer were to place a Student and a SpecialNeedStudent into the ArrayList and randomly retrieve one back (as type Student), the software is faced with the problem of determining if this Student is of type Student or of type SpecialNeedStudent. In such situations, the language must supply a mechanism to determine type at runtime. In Java, this is the instanceOf keyword. Finally, the more subclasses or inheritance deployed, the more difficult it is to read and understand the code. The developer must jump up and down through the class hierarchy to get a handle on behavior and state if they are not implemented directly in the subclass. This is known as the "yo yo problem" (Budd 1997).

E. Subclassing for Specification

Subclassing for specification is the use of inheritance "to guarantee that classes maintain a certain common interface - that is, they implement the same methods" (Budd 1997). In this application of inheritance, the parent class defines in an abstract manner the methods that all subclasses must implement. For example, the student management system contains an abstract class called Person where there are methods defined but not implemented. In Java, these would be methods prefaced with the keyword *abstract*. An example method might be *abstract* getFirstName(). Any subclass of Person must therefore implement this method or itself define it as abstract, which would ultimately render the subclass abstract and uninstantiable. Assume now that the superintendent wanted the ability to reference people by nickname, yet the specifications did not define this behavior or it was assumed. Based on results of the software components developed, this behavior was non-existent. The ability to define a new abstract method at the top of the class hierarchy (in the Person class) named getNickName() would be entirely possible, but this addition would require all subclasses to implement this behavior or become abstract. In essence, the change to the abstract super class renders all subclasses abstract until the new behavior is realized at some level in the hierarchy. The changes in implementation would be easily accommodated, but the possibility of large amounts of additional implementation could become tedious and time consuming if the number of child classes was large. Person might have subclasses Student, Teacher, Counselor, Administrator, PayrollClerk, etc. It becomes apparent this simple addition would result in a large amount of code.

F. Subclassing for Construction

Subclassing for construction is a third application of inheritance (Comp 473 2002). This application might be seen with the use of complex utility methods that could be reused in multiple child classes. For example, one might want to store the image of a Student in a BLOB column in the database. The reading and writing of BLOB columns is somewhat complex depending on database implementation and in Java, JDK/JDBC version (database connectivity). A design would certainly not call for this code to be duplicated in all the different Student types. With that stated, a better design that keeps reuse in mind might call for a superclass that contains the methods writeBlob() and readBlob() and has the name BlobColumnUtility. A Student might be a subclass of BlobColumnUtility with the method of storePicture() and retrievePicture(). Inside these methods Student (and SpecialNeedStudent) would delegate to methods inherited from the BlobColumnUtility. Clearly Student is not a subtype of BlobColumnUtility, it merely facades the utility methods within the utility class and takes advantage of its complex code through reuse. One particular disadvantage in Java is that the developer does not have the ability to inherit from multiple classes. Student may already inherit from Person, which means that either Person must inherit from BlobColumnUtility or vice versa which begins to make the yo yo problem increase in magnitude and readability/understandability decrease. A better solution all together would be to aggregate BlobColumnUtility into Student (also known as Composition) and SpecialNeedStudent or maybe into Person. The methods storePicture() and retrievePicture() could still delegate to methods of BlobColumnUtility, but the difference is that Student would have an internal private reference to this utility class and not inherit from it. Thus, our class hierarchy would remain smaller and thus more readable and understandable.

G. Subclassing for Generalization

Subclassing for Generalization is a form of inheritance where the child class actually overrides some of the behavior of the parent class in order to achieve a more general purpose class (Budd 1997). In other words, a child class is more generic in terms of its ability to behave in a given context. Sometimes this type of inheritance model is unavoidable, especially if one were dealing with COTS (Custom Off-The-Shelf Components) where the developers have no control of code changes or the ability to insert classes somewhere in the middle of the COTS class hierarchy. Lets assume the student management system developers acquired a class library for handling the electronic distribution of grade reports to parents at the end of a semester. This library only handles email distribution, but according to specification, fax distribution was also required but not realized during acceptance testing. The library class DocumentEmailer might have behavior of sendDocument(). A new subclass of DocumentEmailer could be introduced, named DocumentSender which overrides sendDocument() and looks up a preferred document sending mechanism per Student and then either delegates to sendDocument() of the parent class DocumentEmailer or messages a new behavior such as faxDocument() in the DocumentSender class itself. Regardless of which output type was desired DocumentSender is clearly a more general-purpose distribution mechanism for grade reports and therefore was subclasssed for generalization. Immediately one can notice that this is not a subtype simply because the developer has overridden behavior of the superclass. As an example of unexpected outcome, lets assume one is dealing with a distributed system. Student processing might be in some execution path on a server A that has no access to a fax machine connected to server B. If the software message DocumentSender for a family of a Student that requested faxed grade reports while on server A, DocumentSender will surely throw some sort of exception. Therefore, the developers are faced with two costly scenarios. One is the code rework to replace instances of DocumentEmailer with DocumentSender. And two is the retesting and validation that must occur, and in this example, redesign could be in order when validation fails.

H. Subclassing for Extension

Subclassing for Extension "simply adds new methods to those of the parent" class (Budd 1997). Subclassing for extension is very similar to subclassing for specialization, but there is no additional state to maintain. This is important, especially when it comes to testing and validation. Addition of state in combination with behavior could result in a larger number of test cases. For the example in relation to the student management system, lets assume that on some attendance report the superintendent wanted the Student first and last initials (not the entire name) and a total number of honor points. A developer could provide these derivable values (derivable from state already maintained in the Student class) programmatically at report generation time, but lets assume that these code snippets were required on twenty different attendance reports. It might be better to add the methods getInitials() and getHonorPoints() to a subclass of Student named StudentReportDetail. Now the developers are faced once again with the yo-yo problem. Recall that one might have already subclassed Student for SpecialNeedStudent. Where would StudentReportDetail fit into this hierarchy? In Java, the developer can't use multiple inheritance, so from a readability and understandability perspective, there is no good spot for this class in the hierarchy. It is a subtype if it can fit, so the developers do gain the ease of validation and the amount of code addition is minimal. For this example, one might have been better off adding the additional methods directly into the Student class, but again the developers are not always able to have this luxury, especially if the developers are not the owners of the superclass (COTS eliminate this option).

I. Subclassing for Limitation

Subclassing for Limitation is where the use of the subclass is somehow restricted, limiting behavior from the client's point of view (Budd 1997). Unfortunately, the client might not know this until it attempts to message the class that was limited, which would result in a purposely thrown exception from the subclass. With that said, it is clear that this type of subclassing is by no means a subtype and is therefore not substitutable. An example of a use of this type of a subclassing might be in a condition where the Student class had the behavior of getPersonalId(). Suppose the customer was not happy with the fact that this highly secured personal identifier was not locked down as expected. Security could be implemented in a subclass named SecuredStudent by overriding getPersonalId() and ensuring that student's id could only be retrieved in certain contexts. This overridden method would now be implemented to throw an exception. In Java, it might be a custom exception (for this example) or an UnsupportedMethodException. This application of subclassing is clearly not desirable for the same reasons mentioned in other subclassing methods where the subclass in not a subtype. Without substitutability, the developer is forced to go through and revalidate every area of code where the Student object could be an instance SecuredStudent to ensure messages to getPersonalId are catching the exception thrown. As simple as the change may seem in the SecuredStudent class, the testing and rework beyond this class could be extremely expensive to ensure that once the exception is caught, processing continues appropriately. It is the rework beyond the exception that could cost serious time and money.

J. Subclassing for Variance

Subclassing for Variance is where the inheritance between two classes is somewhat arbitrary between parent and child because the two classes share similar state and behavior but do not "possess any hierarchical relationship between the abstract concepts represented by the classes" (Budd 1997). Suppose the student management software has two classes named AttendanceRecord and GradeRecord. Both classes have setRecord(), getRecord(), and getClassId(). From the client's point of view, implementation is somewhat similar. But suppose that getClassId() formatted the class id differently from one class to the next and the superintendent did not expect this behavior on say his or her's attendance and grade reports. The developer decides at this point to keep getClassId() in AttendanceRecord only, and have GradeRecord subclass AttendanceRecord overriding setRecord() and getRecord() because of the differences this type of behavior might require between these classes and inherit without change the AttendanceRecord's getClassId() and its associated member variable(s) to store this information. Consequently, both the AttendanceRecord and GradeRecord (now the child class) have common formatted class id values when a client messages getClassId(). Clearly, the conceptual view of this hierarchy could go both ways (the AttendanceRecord could be the child class instead of the GradeRecord). From the perspective of good design, this example would be scored very low. If the purpose of this type of subclassing is strictly to retrieve common code out of one or the other in hopes to share and reuse, then it clearly hurts the readability and understandability of the code in terms of the hierarchy. In the use or instantiation of these classes, there would probably be no confusion because there would never implicitly be a reference to an AttendanceRecord object in code that was generating a Grade Report. So from a component readability standpoint, this type of subclassing could go unchecked in this example. With that said, this type of subclassing could be a subtype because the language used (Java in this example) would tolerate it, but logically or conceptually there would be no reason to think that subtype or substitutability could be deployed. If a developer put the following line of code in the grade report component, the next developer who needs to support this would probably go looking for a new job:

```
AttendanceRecord gradeRecord = new GradeRecord().
```

K. Subclassing for Combination

Subclassing for Combination is the last common subclassing application to explore. This type of subclassing is more commonly named Multiple Inheritance (Budd 1997). After development is complete the superintendent discovers that the system supports as expected the concepts of Student and Parent, but does not support Students that are also

Parents. Since the classes Student and Parent are available, a new class called ParentStudent could be derived by subclassing from both Student and Parent, thus inheriting state and behavior from both "parent" classes. It appears semantically to be clear that anywhere Student appears in existing code, ParentStudent could be substituted. The same is true for Parent. This type of inheritance would be a subtype application and therefore expedite once again the confidence needed in order to revalidate and move forward after the change in specification. There are two major problems with this approach to subclassing. The first involves the potential for methods that are ambiguously defined from the "parent" classes (Budd 1997). Suppose for example that the classes Student and Parent both had the method named getPhoneNumber(). The behavior of the Student class would be to get the home phone number of the pupil, while the behavior in the Parent class would be to by default get the work phone number of the pupil's parent. So additional rework would be necessary to ensure that when the class ParentStudent is messaged with getPhoneNumber(), the class eliminates this ambiguity by specifying which phone number to retrieve from which "parent" class. Unfortunately, this could be further confused based on the context in which the message is being passed, which in turn would diminish our confidence in our subtype, and increase our vigor in revalidation and ultimately cost for change. The other problem with this type of subclassing resides in the potential for both "parent" classes to derive from a common grandparent (Budd 1997). Suppose the class Person is the "parent" class of Student and Parent. Now suppose that the member variable phoneNumber is in Person. What state will be held for the class ParentStudent for phoneNumber? Will it be the pupil's home phone number of the pupil's parent's work phone number? Now design changes are also necessary to ensure that the software can store the appropriate data at the appropriate level. It may be necessary in this situation to override the getPhoneNumber() routine in the new ParentStudent class or to store locally the phone numbers in the Student and Parent "parent" classes. Somehow, the state must be maintainable for both home and work phone number and the behavior of getPhoneNumber() per context used should be unnoticeable for our subtype promise to hold. Java does not support multiple inheritance. It specifies it through multiple inheritance of interfaces, but implementation of behavior is always at the child class level.

L. Extendibility Conclusion

An OO programming language's extendibility is clearly realized at various levels of complexity (subclassing techniques) and in different ever-changing development scenarios (the examples could be infinite). With inheritance and subclassing being the primary tool for extendibility in OO languages, this paper focused on the OO student management software's ability to change in several scenarios and accomplish to some degree an "extended" software component that matches a change in specification. Soon one would realize that real concerns are not just in the coding changes. Coding changes can be clearly defined based on the examples seen in this paper so far. Although the subclassing coding changes were focused and definable, ramifications of those changes could in some cases go unnoticed if the subclass created was also a subtype, and therefore substitutable. If the subclass was not a subtype, the possible mishandled and unexpected message interactions between the new subclass and existing classes that might still appear to be messaging the original parent class could grow in number. This in turn could bring about faults or unspecified behaviors. Because of the need to look outside the subclass and understand the interaction with other classes more thoroughly, more testing and more work up front to realize these potential problems at design time is required. Ultimately, the cost for change in specification for a software component developed in an OO language is inversely related to the ability of the development team to subclass with care and ensure that subclasses are also subtypes.

VIII. Reuse

A. Introduction

The justifications made for a conversion to the OO methodology (noted in prior sections – faster, better, cheaper), were largely centered on the ability to reuse software assets. The initial focus of reuse was on the code, but over time, experts began to realize and publish that the scope must be expanded to include most of the artifacts developed throughout the software lifecycle. Unfortunately, the expanded scope for reuse presented new obstacles that were vast, complicated and perceived by many to be primarily of a technical nature. However, as this paper will address, over time, organizational and sociological factors gained the attention of management as barriers to a successful reuse strategy. First, the hopes and motivation for reuse will be reviewed. Next, due to a severe lack of case-specific documentation on how reuse strategies did fail in companies, the discussion will focus on the thoroughly documented reasons on how reuse strategy can fail. The reader is encouraged to ascertain the appropriateness of each obstacle based on his or her own experiences. Then, the technical aspects of the problem are revisited, with insight gained from the trials and errors experienced and documented by experts, since the initial efforts of a transition to OO design began. Finally, a review of OO implementation success stories and the applied reuse strategies will be highlighted and related to the reuse obstacle avoidance suggestions noted in this section of the paper.

B. Hopes for Reuse

A common definition of software reuse is the extent to which a component can be used with or without changes in multiple software systems, versions, or implementations (Gamma et al.1994). An understanding of the motivation for software reuse is an appropriate starting point. Consider the promises made for OO, with specific regard to the potential for code reuse.

1. Reduced Coupling

Compared to a procedural environment, a properly implemented set of objects should promote reduced dependencies during compilation. With an optimal hierarchy design, Creational and Behavioral object patterns that support timely substitutions of objects can detach the caller from the logic they intend to use (late-binding). This implicit degree of separation should improve the ability to implement logic extensions in the future without adverse impact on current object subscribers.

2. Complexity Hiding

Encapsulation should improve the coding efficiency of developers, since all that needs to be dealt with is an object's interfaces. Since it has been proven that the human mind can handle limited amounts of information at one time, the additional level of abstraction afforded by OO encapsulation should allow for more complex systems to be built in less time. The intricacies of class implementation should not be pursued and visibility mechanisms (modifiers: public, private, protected, etc) are used to restrict the temptation.

3. Increased Scalability

Centralized, reusable code should lead to a smaller executable footprint. Since dependencies are built only upon the interfaces that link objects together, mutually exclusive compilations will contribute to just-in-time assembly calls. Execution time should be reduced due to optimized message passing through object hierarchies. Examples might include in-memory Singleton or Facade designs, or clever threading schemes.

4. Better Division of work

A class is the core unit of work. Its logic is self-contained with specific pre and post conditions to solidify its contract (function, intended purpose). Developers can be assigned to develop, extend, and maintain portions of the component or system, without adverse impact to other assemblies or program logic. Opportunities for parallel and asynchronous development (with future objects) are increased.

5. Better abstraction

Abstraction will promote better containment of state, behavior and identity by prescribed definition and better separation of the interface from the implementation.

6. Improved time to market

The ability to support new requirements by reusing and extending code from an existing modularized asset base, which has already been thoroughly tested, should yield a significant advantage over traditional procedural approaches. In procedural systems the code seek-time is significant due to the disparate contexts in which it is found and often copied from, contributing to the inconsistency of its intended purpose.

While the motivation for reuse seems concise and attainable when considered exclusively from the other sections of this paper, the reality is that applied reuse strategy has met many challenging and interwoven obstacles. While there is no single formula for reuse success and little to no documentation of actual failed attempts by companies to apply reuse strategy (with clear reasoning behind the failures), there is realistic hope for future success by studying the obstacles (and how they are interrelated) that can lead to reuse strategy failure.

C. Obstacles to Reuse

1. To Reuse or not to Reuse, that is the Question

A fundamental issue, which is often overlooked, is determining whether an organization pursuing a reuse strategy should even consider systematic reuse as a goal. Reuse strategy is more than a buzz phrase to inspire renewal and competitive action; it is a serious, long-term and expensive, yet rewarding commitment. If an organization does not perceive their technology infrastructure as a core competency by which to deliver their primary business strategy, then higher granularity (component or application level, across teams and business units) reuse is unlikely to be achieved (Jacobson, Griss & Jonsson 1997). The ability to present justifications for the organizational restructuring and considerable up-front costs to support systematic reuse, are most easily sold to management in the form of competitive advantage gains, by creating a closely coupled definition of business strategy and reuse strategy. If technology is not a perceived or real solution to maintain and excel competitive advantage in the target organization's market, then reuse should not be a core, organizational strategy. When an organization's market is not technically competitive, reinvention of software is more prevalent than reuse (Schmidt 1999). Furthermore, reuse strategy stakeholders need to remember that reuse is merely an approach by which to achieve important business goals (whether software oriented or not), such as improved quality, flexibility and reduced time-to-market, and should not be considered the primary goal (Williamson 1997c). In certain scenarios reuse may be inappropriate.

2. Active Management of Reuse Strategy

Horizontal management support must be acquired early during the reuse restructuring effort and sustained in the form of active involvement throughout the company's existence. The establishment of reuse cannot be approached like a typical project with a finite timeline. Historically, this involvement tends to be perceived as a achievable by using existing management structure, with the reuse motivation and benefits statistically defined to some degree, and technical training on the mechanics of reuse initiated among developer ranks (Jacobson, Griss & Jonsson 1997). This is due in part to how existing software projects are managed, vertically, against specific departmental or area budgets, with focus on meeting deadlines by delivering good enough solutions. Such narrowly focused, short-term strategies have commonly been referred to as over-engineering avoidance. However, producing components of reusable quality requires management alignment across the organization, since the development effort takes longer and requires better vision and strategy alignment of business requirements and potential design solutions (from class level granularity to application families). A long-term perspective can be a difficult sell since competitiveness is typically measured by short-term advantages. Requirements gathering and development, focused on behalf of a functional area of the organization, leads to designs and components that are not readily available to other projects since specific business functionality has often been embedded at levels inappropriate for reuse (Coatta 2000). While a single developer typically reuses (component 'calls' and not code snippet pasting) only 10 to 20 percent of components or classes they've developed on past projects, initial attempts to promote systematic, organizational reuse will not yield proportional efficiencies across developers, due to the added complexities of managing a reuse organization, which will be discussed throughout the rest of this paper (Jacobson, Griss & Jonsson 1997).

3. Economical Challenges

Additionally, support for corporate-wide reusable assets often has significant economical challenges. Organizations structured as cost-centers can experience difficulty establishing appropriate taxation or charge-back schemes to fund their reuse groups (Schmidt 1999). A centralized reuse support staff's resources must be managed with an unbiased, broad focused perspective. Re-centralizing vertically devoted IT resources, at least from a monetary perspective, is likely to be met with significant managerial resistance (Williamson 1997a). Often, the startup costs prove too extreme for some companies whose markets are more mature and justify less innovative competitiveness (Jacobson, Griss & Jonsson 1997).

4. Administrative Impediments – Cataloging, Search and Discovery Limitations

Managing reusable software assets on a broad scale requires significant policy change and technical enhancements. It is hard to catalog, archive and retrieve reusable assets across multiple business units within a large organization. Code scavenging within an individual team's library is a more common realization than efficient search and implementation of components outside of a developer's immediate workgroup (Schmidt 1999).

While library tools, such as those that support centralized code backup and extraction, have been around a long time, the ever-increasing complexity brought on by OO design methodology requires a much more sophisticated and refined asset storage environment. In addition to the common version control mechanisms available to date in tools such as Microsoft's Visual SourceSafe, developers in a reuse organization will required additional capabilities to support advanced logical searches of assets in the repository extending well beyond attributes such as control author, object name, properties, methods and interface definitions.

Microsoft's .NET Framework, for example, has taken steps to advance the Intellisense documentation capabilities by providing rich XML environments to support in-line documentation (Ndoc – C#), with minimal required effort. However, it is still up to the developer to provide their interpretation of the defined object's intended use. Understanding the breadth of future implementations of their component is rather difficult, bordering on impossible. The metadata will be only as good as the developer's perception upon the object's release or maintenance check-in. Interpretations of class functionality, design pattern implemented, and object dependencies are new requirements of an OO world.

5. Problem Reporting and Version Control

Another extremely important capability is the ability to report problems with a reusable component, remotely, and to broadcast revisions of those objects to all consumers. The inability to synchronize such revisions is a major hurdle to component reuse (Jacobson, Griss & Jonsson 1997). Diverging code bases and sacrificed quality result in lowered trust of component stability and value, which is detrimental to reuse strategy success (Coatta 2000). The Mosaic library is an example of an attempt to deliver these new search and discovery requirements (Poulin & Werkman 1994). Without these advanced OO configuration tools and human support networks, a phenomenon known as write-only libraries becomes prevalent, in which assets are cataloged but never used / referenced directly as intended. Opportunistic code scavenging takes a new form, in which additional assets are checked-in which are only slightly different from the original object. Each new one-off version likely incorporates a domain specific solution into the object, which further dilutes the original intent of the object and consequently diminishes the reusability further. These additional cloned assets create additional overhead for the inquirer to discern, likely causing frustration and mistrust in the reusable asset base, leading to additional one-off development and compounding of the problem.

6. Narrow Focus on Code Reuse Only

Research shows that limiting reuse practices to the coding phase is a common misconception. Since only 10-20 percent of a software project's total cost is absorbed in the coding phase, a major reuse opportunity is forgone (Jacobson, Griss & Jonsson 1997). Further, Jacobson, Griss and Jonsson (1997) emphasize the need to expand the definition of a reusable component to include a packaged deliverable containing the use case models, analysis and requirements documents, and testing standards (including historical results), in addition to the expectation of the implementation model (programming language, classes, interfaces, variation point definitions, etc). The UML unit should encompass (package) this broad range of models as a complete deliverable asset to promote correct reuse of the ultimate deliverable, the code (Jacobson, Griss & Jonsson 1997).

7. Getting Started is Difficult

Starting a reuse strategy is difficult, especially if OO methodology is a new venture in the organization. Possessing a limited asset base, of reusable quality, can inhibit motivation. The key is to start small and build reusable assets incrementally with a horizontal (cross-functional) focus (Hunter 1997). The ability to identify variability and commonality across the domains of an organization requires time and iteration. "Since rarely will an asset's design ever be correct in early releases, management must have the vision and resolve to support the incremental evolution of reusable assets" (Schmidt 1999).

8. Opportunistic Reuse – Convenience and Safety vs. Proactive Design

When dealing with a legacy code base in the critical path of the system functionality the conversion to an OO methodology is often perceived as too dangerous and expensive since significant application segments must be converted together to maintain operation mass. Conversely, when adapting existing OO components for reuse in another system, perceived risk may be indeterminable due to the afore mentioned administrative and organizational (lack of centralized knowledge/asset experts) impediments. The good news is that there is much 'low-hanging fruit', estimated by many to be in the 60-70% commonality range, across business applications (McClure 1995). Microsoft's .NET Framework strategy, as a response to this issue, exposes a Common Language Runtime environment, to encourage leveraging of existing assets (often legacy-based), on a common OO platform, supporting cross language inheritance. Microsoft has marketed the expectation that new web architectures can leverage existing legacy assets (i.e. Cobol or Fortran) to promote reuse within the lower tiers (i.e. Data and Enterprise component layers). However, convenient redeployment of existing code that has not been planned or designed for efficient and flexible reuse should not be misconstrued as a viable Reuse strategy (Hunter 1995).

9. Reorganization to Escape Silo-Oriented Development

The argument could be made that the most commonly noted cases of successful reuse implementations outside of the business applications arena (infrastructure – networking, security, data tier facades, etc.) are simply due to the fact that the OO discipline is still young. The infrastructure models have a naturally centralized, horizontal reach over the functional areas of the organization. Vertical budgets tend to be more vested in the tangible investments that reach internal and external customers in competitive ways. Thus, the natural evolution of opportunistic code scavenging was realized, in the place of well-conceived reusable designs, due to the unique constraints of each business area (deadlines, logic, customized UI, client integration points, etc.). The significant remaining opportunity lies within the intricacies of the business application domains. Consider the common centralized organizational structure that supports and drives the infrastructure systems. Now contrast those with the typically vertically decentralized development teams supporting (often funded 100% by) their respective business application areas. The justification is easily made for the need to redefine the organization's needs at various levels of granularity, from classes to Components, to Application Systems and their final associations as Application families, and then support them through a formal reorganization of staff, policy and continuous management. This multi-level, team-based organization should produce a standards-based software development environment, which is more stable and reusable, similar to other historically proven engineering disciplines (Jacobson, Griss & Jonsson 1997).

10. Psychological Impediments

Attempts to centralize development efforts will not go unchallenged though. Developers may perceive top-down reuse efforts as an indicator of management lacking confidence in their technical capabilities. While it may be argued that the not invented here syndrome, displayed most often by extremely talented developers (or by those trying to gain such recognition), is a sign of insecurity or immaturity, it is likely undeniable that most developers have experienced this historically. Software development remains more an art than science because a developer may often embed their imagination and personal style into the final solution. This psychological impediment, which seems trivial, can prove to be one of the most detrimental roadblocks to systematic reuse (Schmidt 1999).

11. Reuse Measurement & Incentives

One suggestion to motivating developers to look beyond their creative differences is to develop a metrics system based on reuse success indicators and provide incentives to drive behavior and results. This suggestion has been met with much debate. Proponents of this approach stress that the measurements should target processes and teams, not individual developers, to avoid afore mentioned psychological impediments. Further, rewards should be tied to the actual, consistent reuse of the components they develop (Williamson 1997b). This approach reduces the ambiguity of measuring the reusability quality level of a component. However, opponents to the idea may argue that at the end of the day an individual developer's career and long-term success is tied closer to actual results. Until the quality of the reusable asset base reaches a meaningful breadth and sustains quality, proving its validity, resistance is likely to continue.

12. Surface vs. Structural Similarity – Expertise Level

Additionally, studies show that more experienced developers handle surface similarity (compared to problem domain) decisions better than novices, choosing to reuse less often, which is contradictory to reuse metrics based solely on reuse frequency (Irwin & Monarchi 1996). Thus, library search facilities that perform only keyword searches are prone to improper reuse if they are in the hands of an inexperienced staff. It is important to note however, that the Irwin and Monarchi study (1996) showed that most experience levels reused components appropriately, based on structural similarities. However, structural knowledge is not readily distributed with most source control tools available today. Acquiring structural awareness should be limited to an object's interface. Attempts to expose the implementation details of classes or components violate the motivation (argument) for inheritance and encapsulation (information hiding).

Providing numbers for one proponent's claim (Williamson 1997a), management could expect a 30 percent growth in reusable quality components when a 5 percent royalty is offered to those who author the components that have proven sustained reuse. Further, a 15 percent royalty can yield 50 percent faster growth in the component library. "Encouraging reuse requires lots of carrots and sticks in organizations where reuse is a relatively new practice. But once it becomes systematic, incentives become unnecessary; it's just the way things are done." (Williamson 1997a). Some experts consider this bribe ineffectual since the strategy is often misconceived (Jacobson, Griss & Jonsson 1997). However, a total lack of incentives can be a deterrent to reuse as well.

13. Expressing Management's Commitment to Reuse Strategy & Accountability for ROI

Reuse measuring is important to realizing effective reuse, whether an incentive plan is associated with the metrics or not. Management must have an indicator of where they have been, measurable goals, and an indicator of current direction in order to adjust strategy, management style and focus to ensure continuous progress. However, it is imperative that the results be shared in a timely fashion, with those who are driving the results. Management's feedback is an indicator of their commitment to the strategy and can significantly influence the adoption rate of a reuse program. Finally, action must be taken against the metric results, including but not limited to, capital investment adjustments, additional training of developers and analysts, and deficiency corrections (Jacobson, Griss & Jonsson 1997). Accountability for a return-on-investment against reuse initiatives is critical. How ROI is determined has proven to be a controversial topic. Two interesting formulas follow (Jacobson, Griss & Jonsson 1997).

Relative Cost due to reuse of components

$ROI_{saved} = C_{saved} / C_{no\text{-}reuse} = R * (1 - F_{use})$

Where:

R = Total Size of Reused Components / Size of Application System

Fuse = Relative cost to reuse a component (rather than developing from scratch) commonly 20% incremental cost, with varying opinions suggesting a range between 3% and 40% (although this is the subject of great debate, when factoring in all phases of the development lifecycle)

So, if R = 50%, the estimated cost to deploy a system with reused components would mean only 40% of the development cost is avoided (theoretically).

ROI to develop the set of components

ROI = $C_{family-saved}/C_{component-systems}=(n*R*(1–F_{use})–R*F_{create)/}$ $R*F_{create}$

Where:

n = number of candidate application systems

Fcreate = Relative cost to create and manage a reusable component system (suggested between 1 and 2.5 times the cost of a system without reuse). Since this cost is usually much greater than Fuse, must reuse a component several times in several systems to make the math worthwhile (which seems logical).

So, if Fuse = 20% and Fcreate = 150%, the breakeven point would be > 2 application systems (n) to warrant consideration of a reuse strategy.

(n * .8 – 1.5) / 1.5

Additionally, Hunter (1997, p. 1) provides an interesting perspective on calculating ROI, using a less formula-regimented and more financially driven approach.

> For a midsize to large application development organization (that is, a shop with 100 to 500 developers) that already has a reuse methodology, the first-year cost of a reuse program may range from \$535,000 to \$1.15 million, depending on the approach to catalog implementation and incentives. Subsequent annual expenditures will range from \$485,000 to \$795,000, not including costs for analysis and identification of reuse opportunities within the development life cycle. The cost of such activities will vary with the complexity of the project. Therefore, to be even minimally profitable, a reuse program for a midsize to large organization must return well over half a million dollars per year.

Although it takes some doing, the return on investment can be calculated. Assuming a developer is paid \$75,000, a company's reuse program must offset the full-time efforts of seven developers or, in a shop with 100 developers, yield a 7 percent productivity improvement to achieve the minimum required return of \$500,000. Based on Gartner Group estimates that consistent use of a reuse methodology increases application development productivity by 30 percent, it is reasonable to predict that a reuse program that includes all required elements will produce an ongoing annual ROI equal to or greater than 200 percent within two years for a shop with 100 developers.

14. Reuse Metrics

Other metrics often considered include (but not limited to) (Sultanoolu 1998):

- Avg. number of methods per class
    - (-) Increases testing complexity
    - (-) Extensibility reduction as this number increases
    - (+) May promote more efficient reuse however; more shallow hierarchies with functionality addressed early on
- Inheritance Dependency depth
    - (+) Tree Depth preferred over breadth in terms of reusability via inheritance
    - (-) Deeper trees increase testing complexity
- Degree of object Coupling
    - (-) Prevents modularity and reuse
    - (-) Limits extensibility
- Method Inheritance Factor
    - Ratio of total inherited methods to total available methods
    - Measuring design effectiveness and efficiency of message passing through the hierarchy
- Attribute Inheritance Factor (same motivation as previous)
- Many others related to Modularity, Testing, and Extensibility (Yazylymn, yunus.hun.edu)
- Metrics related to development duration and Maintenance
    - Effort and Duration in terms of workweeks, months, etc
    - Time to Failure
    - Time to Recovery (from point of failure)
    - Suitability – change requests per object count in application
    - Many others

15. Knowing When to Outsource

Arguably, the ultimate form of reuse may be to outsource the effort completely. Of course, the integration (custom code, testing, etc) issues often remain since components or application frameworks intend to serve as a starting point towards a context sensitive solution.

Component Based Development (CBD) appears to be the next (approx. est. 1993) level of abstraction. A component is essentially a series of classes/objects that are cohesively grouped to perform a generic, yet meaningful set of tasks. A facade (design pattern object) exposes only the required interfaces to promote efficient, effective and decoupled use of the component. Facades also promote backward compatibility (multiple façade versions are allowed in same component) by allowing only indirect access to the underlying objects and functionality (Jacobson, Griss & Jonsson 1997). Proponents thus argue that reuse is optimized. Some opponents warn that the complexity, testing and various other flaws encountered with OO development (noted in this paper) are worsened and that CBD represents an attempt to solve reuse strategy failure by promoting responsibility for success above the developer ranks. Both viewpoints carry some validity, depending on the maturity (staff organization and asset base) of the reuse organization and decision making involvement among the various levels.

Research shows that many companies are shifting their strategy to CBD and integration efforts, rather than complete in-house development, in order reduce costs and time-to-market. Potentially, only a company's most vital, unique core competencies are retained for custom in-house development. Brian Morrow, director of CBD at Plano, Texas-based Texas Instruments Inc. is quoted by Williamson (1997c, p1) stating that "today we see the off-the-shelf components selling at between one-third and one-fifth the cost of developing your own." While cost reduction is one motivation, the issue of context specific business rule integration remains a challenge. Proponents suggest that companies with a mature CBD environment experience only 5 percent variation in business logic requiring customization (Williamson 1997a). Opponents are eager to remind lower management and analysts that the sale of CBD is most often made at higher levels of management (who are less privy to the true technical issues surrounding this higher level of abstract development) due to the immediate cost implications and long-term contracts that must be approved. Further, they argue that likely conflict between the actual integration of CBD, which must occur 'in the trenches', and the CBD liaisons requires constant monitoring and negotiation among the various levels of the reuse organization. This obviously adds to the previously noted management complexity issues. CBD strategy raises new issues (of unknown impact) that may hinder reuse strategy further. ROI calculations and reuse metrics, historically used to assist in make vs. buy determination, cannot solely justify a strategic move to CBD.

16. Skill Levels Required of OO Analysis & Development

Finally, a critical barrier to reuse strategy success is the inability to recognize what has high potential for reuse, which implies a higher level of developer expertise due to the complexities introduced by OO development. A combination of education and training in OO design theory, more heavily weighted by substantial experience in building OO solutions, is essential at all levels of the development team. Assessing a developer's ability to conceive OO designs and then implement them is a difficult distinction for managers (especially when the managers are OO novices as well) to make. Acquisition of analysis and design pattern awareness is essential to solid design conducive to successful reuse (Schmidt 1996). Experience implementing the patterns and architectures solidifies the developer's design choices and efficiency. Conversely, research has shown that novice developers tend to focus on attempting to reuse too much.

Design Pattern knowledge could be equated to the rules that guide a particular spoken language's constructs. Understanding the pronunciation and definition of a decent amount of words does not imply that a person can fluently convey ideas verbally or in writing in that language. Coding language fluency (cohesion, coupling, substitutability, etc.) is vital to software reusability.

D. Design Patterns Bridge the Technical Gap

The ability to communicate design concepts and logical reasoning is critical to a reuse strategy. A shared descriptive language will promote proper ongoing implementation of assets targeted for reuse. Design Patterns assist software reuse productivity.

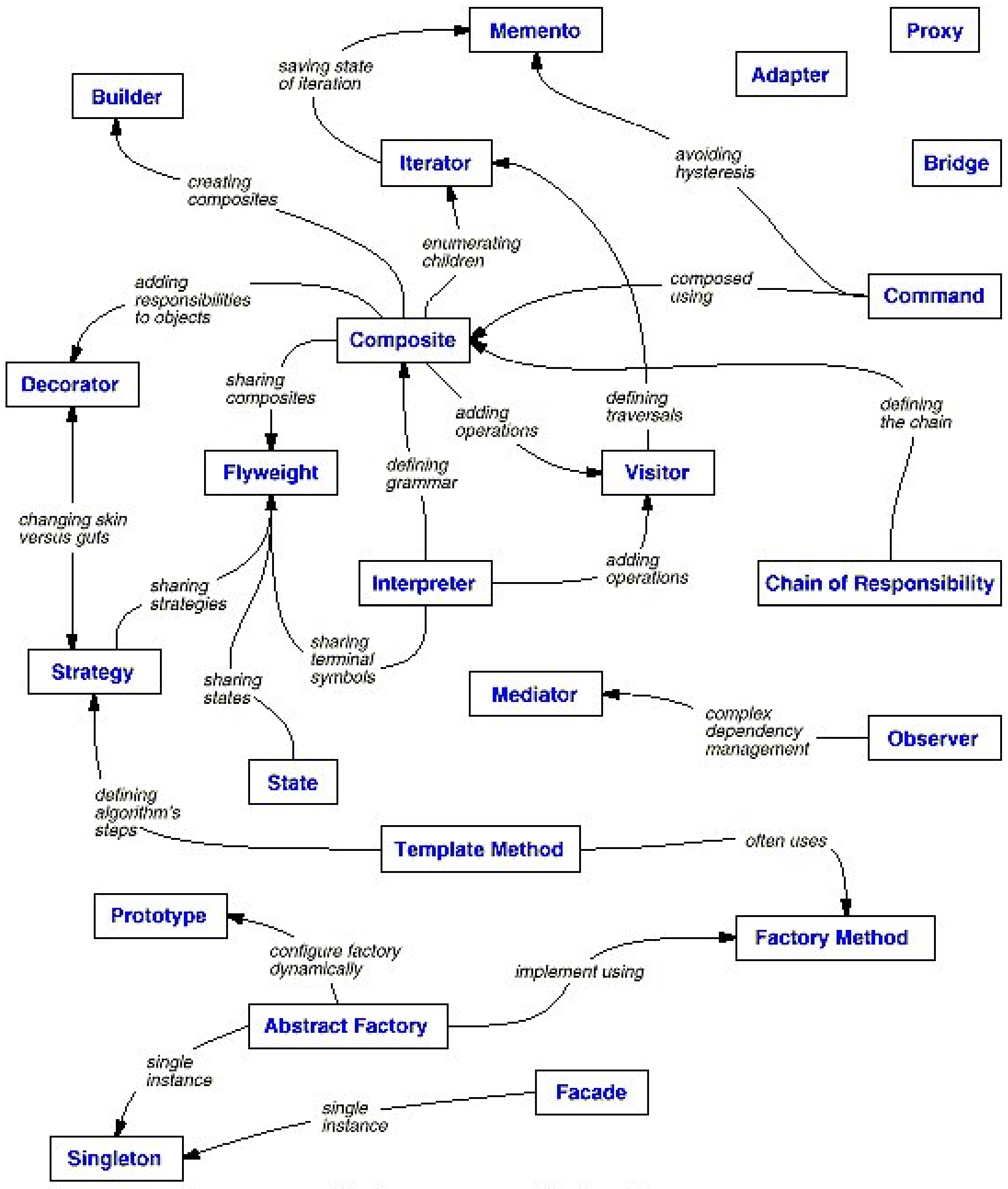
A pattern is a recurring solution to a standard problem. To that extent, patterns are nothing new since they essentially capture practical experience acquired by expert developers (Schmidt 1996). A Pattern definition contains (Gamma et al.1994):

- Common Name – The basis of software design vocabulary; allows developers to architect software at higher levels of abstraction. Should be part of a library's nomenclature, to assist in efficient and correct interpretation of developed assets.
- Problem – Describes the problem and its context, along with descriptions of class or object structures that are symptomatic of an inflexible design. May also include pre conditions to justify use of the pattern.
- Solution – An abstract description of a design problem and how a general arrange of classes or objects solves it.
- Consequences – The results and trade-offs (space, time, flexibility, extensibility, and portability) of applying the pattern to assist in evaluation and selection of the chosen pattern.

Pattern family names and participant cardinality vary slightly by source, but in general, they fit into one of three categories (Gamma et al.1994).

- Creational – Abstract the instantiation process. Become more important as systems rely more on Composition in the place of inheritance.
- Structural – Describe ways to dynamically (at run-time) compose interfaces or implementations using inheritance, to realize new objects with more specific functionality.
- Behavioral – Describe patterns of communication between objects or classes that are difficult to follow at run-time, allowing the developer to concentrate on the ways they are interconnected instead.

The following figure aims to illustrate design pattern interconnectivity and expressive capabilities.



Design pattern relationships
(Gamma et al. 1994)

Patterns are not intended to replace developer creativity. They are a vocabulary by which to communicate a means to a successful design output, of which there are many possible solutions depending on the problem domain context. Pattern awareness and implementation experience provide the developer community a forum for expressing design ideas in a consistent, unbiased and depersonalized manner. The previously mentioned sociological issues associated with software reuse may be tempered over time as the developer community's pattern awareness matures along with the OO discipline.

The intent of this portion of the paper is merely raise awareness (or remind) to this facet of software development. Without the ability to communicate technical ideas effectively the developer community will struggle to convert from a procedural world to an OO methodology. Research suggests that lack of pattern awareness or inconsistent and improper use accounts for a significant portion of the reason for OO's failed promise of reuse from a technical perspective.

E. Successful Implementation Examples

An understanding of what the hopes for software reuse were (and still are), and the obstacles that developers and management must consciously and continuously manage against, is likely the best defense against a reuse strategy failure. This self-proclaimed deduction is based on the fact that extensive research has yielded little to no credit worthy evidence of why reuse strategies have failed in actual implementation (from a company's viewpoint). Organizations are apparently less likely to share (publicly anyway) examples of their strategy failures and the reasons that led to them. This may be due in part to the complexity of the issue. The array of factors that can negatively impact a reuse effort, hopefully, has been outlined well in this document.

However, there is no shortage of success stories (claims?). As a result, those interested in developing a reuse strategy must draw from the experience of former analysts, developers and managers who have experienced the failures first hand. A couple of the more interesting cited success stories follow (many more examples can be found in the bibliography). Each example addresses one or more of the issues noted in the obstacles section of this paper, as highlighted in italics.

> Household International in Northbrook, IL developed a meta-model of its current environment called the Logical Overview System (LOVS), to be the baseline for managing an initiative to build reusable routines and migrate to client/server. Having a portable library of reusable routines controlled through LOVS helped avoid building new routines from

> scratch. Estimated 1993 savings from this initiative were over $3 million. Development of 435 new modules was avoided last year, and a new system was delivered 132 days faster than if LOVS was not in place (McClure 1995)
>
> US Naval Surface Warfare Center, Virginia Beach, VA built a common architecture for a family of combat direction systems. On fourteen ship system upgrades, the center achieved a level of reuse of 89-99%, a 3-fold reduction in defects, and an 8 to 10 fold increase in productivity (McClure 1995).
>
> In a pilot project involving compiler and compiler-tool test suites at Motorola, an 85 percent reuse level and a 10:1 savings ratio have been demonstrated. The primary reusable components are design and documentation (McClure 1995).

A 55-member staff at General Accidental Insurance Inc. is replacing the company's legacy systems with client/server applications, one line of business at a time. Reusability and component-based technology are key to the effort. Estimates show that Component-Based Development enables the organization to deliver systems three to four times faster than was possible with applications written in Cobol. The project's keystone is a reusable-rating engine. Historically, the theory and concepts were reused, but never the code. It took four tries to build a reusable-rating engine that can accept different algorithms for different policies, but the entire effort took less than a year (Williamson 1997a).

F. Conclusion on Reuse

Considering the promises made by the potential for software reuse, the complexities involved with managing the strategy and how young the OO discipline is in general, it is difficult to say whether the promise of reuse has failed. To date, many organizations have likely not optimally managed the organizational, architectural, and human resource issues involved to achieve reuse strategies. However, as the OO discipline matures (as well as the development community's skills) and a decent history of measurement has materialized, a final opinion may be rendered. A solid understanding of an organization's core competencies, along with how and when software reuse can / should be leveraged to maintain a competitive advantage remains an essential driver of successful implementation. Measuring reuse results and shortening the feedback cycles will aid in the adoption of and adaptation to this new (from an adoption rate viewpoint anyway) discipline. Compared to the procedural development environment, reuse, considered exclusively from the other issues noted in this paper, continues to offer significant advantages. However, with organizational issues resolved, the technical learning curve remains a formidable challenge and only time will prove whether it can be surmounted.

IX. Testing and Debugging Object Oriented Systems

A. Introduction

For many years developers have used a waterfall approach to procedural system development wherein testing has evolved into a well-defined set of processes. Different testing approaches (functional and structural) with various methods (boundary value, path, equivalence class, etc.) are applied to various scopes of system detail (unit, integration, system and regression) to identify test cases. With the advent of OO systems these tried and true testing processes were all that was available for testing OO systems. (Note: Recent works on different techniques specific to testing OO systems have been proposed; however, from a historical perspective, when OO systems were developed, procedural testing processes were all that were available). Even though there are fundamental differences between procedural and OO system development, many people theorized that some of the benefits OO design provides, such as naturalness, better modeling and reuse, would apply to the testing phase. These benefits, combined with OO specific features, such as inheritance, polymorphism and encapsulation, would reduce the error density and necessary testing so OO systems would be easier and therefore less costly to test than procedural systems. It is natural to deduce that if OO systems were better designed, better coded, and more reliable they should be easier to test than procedural systems. What has come to light is that the inefficiencies of testing OO systems far outweigh any efficiency that was realized and has caused an increase in testing complexity and cost when compared to procedural systems. “There are several aspects of object oriented programming that makes it easier to conceptualize, maintain and reuse but those very conveniences also have a cost in terms of testing” (Beierle 2001).

This section will start with a review of several approaches and levels of testing comparing their use in OO and procedural systems. Next, the reasoning behind the hopes for easier testing in an OO environment is explored. Finally, the added testing obstacles inherent to OO systems are discussed.

B. The Testing Process

The premise for testing OO systems is identical to the premise for testing procedural systems--to make sure the system should function as it was intended without any additional, undesired functionality. Testing is typically broken into two processes. The first process is to verify that the code did its intended job, which helps establish confidence. This is sometimes referred to as sunny-day testing because only correct situations are tested--the program is running in a perfect world. The second process, once confidence has been established, is to find situations where the code does not work. This is sometimes referred to as cloudy-day testing because this process tries to find out if the code does anything it is not intend it to do by using invalid input or out of boundary situations--the program is running in the real world.

1. Functional and Structural Testing

To test systems two approaches are used: functional and structural testing. Functional (a.k.a. black box) testing checks that the correct output is derived from a specific input without regard to how the code made it happen (conversely, that an incorrect output is not derived from a specific input). Structural (a.k.a. white box) testing is used to examine the code's details and ensure all situations in the code have been checked because there may be some scenarios that occur in the code that are not identified by functional tests. Neither approach, taken exclusively, is sufficient (Jorgensen 1995). Therefore, it is typical that test plans include both functional and structural test cases. The approach to functional and structural testing is generally the same in procedural and OO systems because they use the same processes to identify test cases. The differences become more apparent when the two approaches are applied to the various levels of system detail.

2. Levels of Testing

The methods used for functional and structural testing processes are applied to three levels of detail within the system--unit, integration and system. Testing begins at the unit level; however there is some debate as to what constitutes a unit in OO systems. "There is nearly universal agreement that the class is the natural unit for test case design. Methods are meaningless apart from their class" (Binder 1996, p.1). Even though the class is considered a unit, the methods within the class are tested individually so traditional functional and structural testing techniques are fully applicable at this level (Jorgensen & Erickson 1994). A class in an OO program can be compared to a procedure or subroutine in a procedural program, where the methods within the class are similar to the functions within a procedure or subroutine. However, there is a difference in that a unit in an OO system is a more complicated structure than a unit in a procedural system because the unit (a class) must deal with the combinations of encapsulated data, inherited data, methods and object state. After unit testing, integration testing begins. "Integration testing is the least well understood of the three levels" (Jorgensen & Erickson 1994, p. 32). At the integration level of OO systems the units (classes) that interact are tested. This can be compared to integration testing in procedural systems where related programs or modules are tested together. Once again, the fundamental differences between class structure and interactions in OO systems and program structures and interactions in procedural systems complicate the identification of integration test cases for OO systems. The interaction of classes within an OO system tends to bring the integration test closer to the unit level than it does in a procedural system. This sometimes results in unit testing and integration testing to be confused as one in the same. System testing is the final level and is similar for procedural and OO systems. At this level all the components or classes are put together and tested as a whole. Alternatively, regression testing is a special level which deals with re-testing existing applications due to changing code from a maintenance standpoint (either for an enhancement or fix) to make sure existing features still function properly. The highly incremental and iterative development cycle of OO systems blurs the distinction made between integration and regression testing that is clearer in procedural systems (Winter 1998). The reason for the blurred distinction lies in the fact that when classes are modified during one or more of the development iterations, a regression test is necessary because interacting classes that have already been tested must be re-tested to check if they react properly to the modified class.

3. Definition of a Bug

There are various terms used to describe the problems that are found during testing - bugs, errors, faults, and failures to name a few. As a practical definition a 'fault' is the result of an error in the code (Jorgensen 1995). It seems that OO system developers prefer to use the term 'bug' instead of 'fault' so 'bug' is the term that will be used in this section.

C. Hopes for Easier Testing

As stated in previous sections, the OO paradigm hoped to provide some time and cost reductions during the early stages of system development that would lead to more reductions during the testing phase. The hope was that the OO development process utilizing reuse, iterative development, and encapsulation would produce fewer lines of code that would reduce testing because there would be fewer errors.

1. Reuse

The construction of classes facilitates the reuse of objects in an OO system. A class that represents a specific object, such as customer, can be reused in many different applications. Since OO systems are intended to utilize reuse, the hope for reduced testing lies in the assumption that by reusing existing classes, less new code would be written and thus, there would be less to test. "In addition to speeding development time, proper class construction and reuse results in far fewer lines of code which translates to less bugs and lower maintenance costs" (Montlick 1999). In procedural development there typically is not a lot of reuse when compared to OO systems. Many developers of procedural systems either reuse sections of code copying them from program to program or call a program stub that performs some common function, such as date manipulation. Even though tested code was copied it is almost always modified in some way to fit the new application; therefore it must be tested again. A selling point for OO systems was that classes were either used as-is, so no re-testing was necessary or if additional methods were required the class was just extended to a subclass so only the new methods of the subclass would need to be tested.

2. Iterative Development

OO systems are naturally suited for iterative development due to the independent nature of classes that can be evolved as methods are added or modified. Each class is naturally independent in the sense that it can function on its own. This independent nature allows developers to write small test programs to test the class after each modification (Kim & Wu 1996) which leads to the iterative design-code-test cycle. The hope was the iterative development of the class would allow developers to find and reduce the number of bugs earlier in the development process (Binder 2001). This process would also strengthen the link between testing and design because every change would need to be traced back to the requirements in the design model. If the change involved changing requirements, then the model must be updated to reflect the changes. In a waterfall development process testing is typically performed after coding is complete, so bugs are not found as early in the development process and they are more costly to fix.

3. Encapsulation

Once a class is coded and tested it is placed into a class library for use by other developers. At this point the code details within the class are hidden from the developer because there should not be any concern about how the class works, only that a method can be invoked to return some result. The hope was encapsulation would reduce testing due to fewer misunderstandings about the class methods since developers would not attempt to read the code to figure out what the methods were doing. The developers would be forced to trust the class because it was guaranteed to work. Encapsulation is the WYSIWYG (What You See Is What You Get) of OO systems. All that is seen is what the class methods can accept as input and return as output, nothing is left to interpretation or assumption so not attempting invalid test cases reduces the overall testing time.

D. Obstacles to Testing

When obstacles to testing OO systems are discussed the problems that the OO specific features of inheritance, encapsulation and polymorphism create is usually the focus. "The combination of polymorphism, inheritance and encapsulation are unique to object-oriented languages, presenting opportunities for error that do not exist in conventional languages" (Binder 2001). These are areas of concern when it comes to testing, but some other complexities OO systems have over procedural systems are also of concern. Differences such as decentralized code, test case identification and raising exceptions also add to the complexity.

1. Decentralized Code

With procedural systems developers are accustomed to testing programs where all the functions and procedures are centralized. Locating the source of a bug does not require us to look outside the walls of the program. The same functionality that was in a single procedural program is broken out into many smaller classes in an OO system. It can be argued that smaller classes ease the process of locating the source of a bug because smaller chunks of code are examined. However, the side effect is that the code is no longer centralized so finding the source of a bug requires looking in many places. This can be a daunting task if the classes are heavily dependent (classes calling classes calling classes, etc.) or if the source code is split up and stored in a complicated directory structure. The use of smaller classes in OO systems also raises the degree to which those classes must interact to produce an intended result. This higher degree of interaction as adds to the testing complexity because there is a higher probability of interfacing errors.

2. Exceptions

Raising exceptions in OO programming languages, Java for example, adds complexity to the testing process because each statement could potentially cause one or more exceptions to be raised (Berard 2002). The handling of these exceptions and how they are thrown and caught can cause branches in the code that would be difficult to test. For example, a class call structure may have four classes in the stack; one class calls another that calls another, etc. If a method in the bottom class in the stack encounters an error it may throw an exception up the call stack until it is handled. When the bug is reported it appears to come from the top class in the stack, because that is where it was caught and handled, so it is difficult to tell that it actually occurred in the lowest class of the call stack. This problem can even occur within a class having private methods that are called from other methods within the class, but finding the source of the bug is a little less daunting. Proper exception handling at the appropriate level can reduce this problem.

3. Test Case Identification

OO systems and programming languages are fundamentally different than procedural systems and programming languages so the methods used for testing procedural systems will not always be an exact fit for OO systems. Encapsulation presents a problem for creating test cases because it is difficult to understand the object interactions (Beierle 2001). The result of encapsulating several methods within a class causes the unit test for an OO system to become more complex than a procedural system. This is because a class as a testable unit essentially wraps together what would have been many unit testable subprograms in a procedural program (Berard 2002). Integration testing for OO systems takes on a different meaning than it did for procedural systems. For procedural systems once the individual units were tested they would be combined with other units one at a time and the interactions between the units were tested. For OO systems integrating and testing classes one at a time may not be an option because a particular object may call methods located in multiple objects to obtain a specific state. Developing a test plan and identifying test cases to deal with all the integration scenarios can prove difficult. Test case identification for OO systems must also deal with the utilization of different modeling diagrams than those used to design procedural systems. Some hope lies in the use of various UML diagrams, especially collaborative diagrams, to more accurately describe the system interactions and identify test cases. "UML collaboration diagrams represent a significant opportunity for testing because they precisely describe how the functions the software provides are connected in a form that can be easily manipulated by automated means" (Abdurazik & Offutt 2000).

4. Encapsulation

Encapsulation and information hiding add complexity on two levels. First, classes wrap methods, data variables and exceptions together and second, provide limited visibility to these items. Testing OO systems depends on verifying the interaction between objects and the state of an object after a method is invoked. To check the state of an object, it is necessary to check the values of its instance variables. This is difficult without knowledge of the variables or how they are modified by the called methods because of information hiding. This is not insurmountable. The use of debugging tools can give visibility to the variables in an object or built-in inherited state reporting methods can be used (Binder 1994). However, the added overhead of configuring the tool for the test or coding the reporting methods is still a reality. Encapsulating several methods in a class can yield a high degree of cohesion between the methods; hence, it is difficult to test each method in isolation. This adds to the testing complexity because the unit test of a method has taken on the form of a more complex integration test between two or more methods.

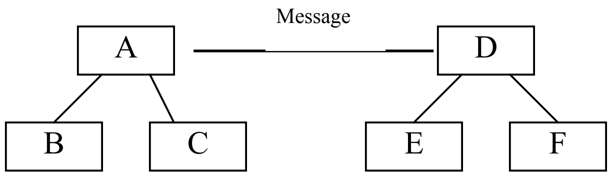
5. Inheritance

Inheritance adds complexity to testing because it is not safe to assume that the method that was tested in the context of the parent class functions correctly in the context of the child class. “Contrary to some hopes, retesting of inherited methods will be the rule rather than the exception” (Binder 1996, p. 2). For example, if a parent class named “Dog” has a method called “bark” and that method is inherited in the subclasses named “Poodle” and “Doberman”, then the “bark” method must be tested in each of the subclasses where it is implemented. This is necessary because the “bark” method will return a different value depending on the context of the subclass where it is called. For example, Poodle.bark will return "yip" and Doberman.bark will return "woof". It is not sufficient to test the "bark" method only in the context of the "Dog" class. This adds to the amount of testing because every time a method is added or modified in a parent class all subclasses must retest the method. The deeper and wider the inheritance hierarchy becomes the more difficult it is to test because the inherited methods get further and further away from the original definition. With inheritance many more test cases must be determined because of the anti-extensionality and anti-composition axioms stated by Beierle (2001) to be:

- Anti-extensionality Axiom - A test set that adequately tests one program will not necessarily adequately test a similar program.
- Anti-composition Axiom - When each component of a program has been adequately tested individually, it does not imply that the program overall has been adequately tested.

6. Polymorphism

Polymorphism adds complexity to testing OO programs because dynamic binding increases the number of separate tests that must be performed and it may be hard to find all such bindings (Binder 1996). Each time a new class is extended from a base class it adds to the testing complexity because it must be tested as a substitution for the base class at runtime. An example from Caspersen, Madsen & Skov (2001) helps to illustrate this point.



Subclasses B and C are possible polymorphic substitutions for class A at runtime, and likewise class E and F for class D. Instead of one set of classes to test, now nine sets of interacting classes has to be considered to cover a single message between two objects. From this simple example it is clear that techniques for reducing the number of test cases are needed.

E. Conclusions

Looking back at the hopes for easier and cheaper testing for systems developed in an OO environment, the use of independent classes and iterative development are the only areas that came close to living up to the promise. In OO development testing can begin earlier in the process because classes are generally small so they can be coded quickly and they are independent so they can be tested quickly. In procedural systems it is typical to code an entire program and then test. It is not difficult to recognize that the ability to develop and test classes incrementally should provide cost savings during development because bugs will be found earlier in the development process where they are cheaper to fix. But these savings quickly disappear because of all the extra complexities that OO systems present compared to procedural developed systems. The hopes for easier and cheaper testing were not fully realized because inheritance, polymorphism, exception raising and the distributed nature of OO systems added complexities to testing that are not dealt with in procedural systems. These complexities, and the lack of many proven testing techniques to handle them, increase the overall cost of testing OO systems when compared to procedural systems.

X. Reduced Maintenance

A. Introduction

After the code has been written, the testing has been completed, and the program has been implemented, a different phase of the software life cycle is entered. In this phase, any work performed on the original application is considered maintenance. Traditionally, maintenance has been a large portion of the total cost of an application, throughout its lifetime, even though it is not directly associated with the development costs. Regardless of the expense classification, the result is the same: program maintenance requires the allocation of a large application budget, and the longer a piece of software is kept in operation the greater the proportion of total expense is maintenance related. Although maintenance is not a profitable service, in any business, it is easy to see that reducing expenses can have a large impact on the bottom line. With maintenance costs skyrocketing, it was only natural to look for a way to decrease the time and money that this phase of the software life-cycle required.

B. Definitions

Before delving into specific details of Object-Oriented (OO) application maintenance advantages and disadvantages, it is necessary to establish a definition of maintenance in reference to a piece of software. Maintenance is defined here as the modification of a software product after delivery to correct faults, improve performance or other attributes, or to adapt the product to a changed environment (IEEE 1983). This definition does not encompass iterative application development.

C. Accountability

Originally, the OO paradigm boasted a decrease in the resources required for application maintenance. This claim was mostly based on attributes associated with the OO program structure. First, and perhaps the weakest argument, is that this structure creates a certain level of accountability that was to some extent lacking in the procedural language structure. The underlying premise of OO programming is the “small module of code” or class. This makes for very easy distribution of work. This also yields easy identification of the programmer responsible for a failure. This system of accountability generates a strong incentive for programmers to do the job correctly the first time; thereby reducing the resources required to fix an error when the application is in the operational phase. The actual savings this level of accountability can be credited with is not easy to determine; in fact, this argument is rarely discussed in the literature and is perhaps best left in the realm of psychology as a control group for scientific analysis is not possible.

D. Encapsulation vs. Complexity

The second potential cost saving attribute is encapsulation. Encapsulation allows the original programmer to put all the code related to an object in one place. Once the program is in the operational phase, if an error is found, it should be easy to identify the actors who are participating in the generation of the error. From this point, a maintenance programmer should be able to narrow down the possible location of the faulty code to a very small fraction of the total source code [See Naturalness and Testing]. In theory, this could potentially reduce the amount of time required to correct an error significantly; hence reducing overall maintenance costs. (Hatton 1998) In practice, research shows that the complexity of an OO application is much greater than a procedural application, thereby making error identification extremely difficult. It is often found that the "failure is a long way away from the fault that caused it." (Hatton 1998) For example, the government facilities at Cheyenne Mountain currently have an OO application in use which promised faster and easier updates than previously implemented procedural applications. At this time, it is believed that due to the complexity of the message passing (which create a virtually untraceable path) in the OO application, maintenance expenditures will rapidly exceed those of preceding applications. (Card & Emam & Scalzo 2001)

Taking into account that one can no longer narrow down the location of the fault, all code is now fair game. Given that the average OO application has significantly more (t=14.67, p<0.0005, α=0.05 (calculation based on original author data)) lines of code (non-blank & non-comment) than its procedural counterpart, the maintenance programmer must sift through more code to find the fault. This translates into an increase in required resources. (Hatton 1998)

E. Documentation

Besides the obvious message passing complexity and overall length of code, it is commonplace to find an OO application which has little to no source code documentation and hundreds of interrelated classes making it very difficult for the maintenance programmer. [See Complexity and Testing] In addition, maintenance programmers are often attempting to fix an application for which an accurate, complete model does not exist. (Fontoura 2002) This lack of a decent model can stem from poor team skills during the initial development phase or from the failure of previous maintenance programmers to update the model with new information. This makes it very difficult for future updates, in much the same way that adding onto a house is difficult without a blueprint. [See Modeling] It is interesting to note however, that if the models are updated with every application revision, a huge amount of effort will be required. This leads to an ironic loop where updating the model decreases the resources required to implement a change, but increases the resources needed for making the model and conversely, not maintaining and testing the model will decrease the resources needed for updating the model, but increase the resources needed to implement a change.

F. Modularity and Testing

Another aspect of the OO paradigm, modularity, creates a standard interface for all input to and output from an application component. Using a standard interface provides a level of abstraction between the data and the functional aspect of an application. When maintenance is required, the programmer should be able to concentrate all efforts into fixing the fault without worrying about inadvertent creation of errors in other components. This should lead to a decrease in required resources for maintenance as well as a decrease in the resources required for regression testing. (Post 2002) The overall quality of an application should also be higher in an OO application as a result of its independent module structure. Whereas anytime maintenance work is done on a procedural application the programmer runs the risk of introducing a defect elsewhere in previously working code, in an Object Oriented application, that risk should be reduced to the confines of the maintained module. (Fayad & Douglas 1997) In practice, it has been found that regression testing falls into the same holes that were dug in the original testing phase. Since the first pass at testing was much more difficult for OO languages, so too is regression testing. This leads to more resource consumption by OO maintenance than that consumed by procedural language maintenance.

G. Modularity and Reuse

Perhaps the largest maintenance cost reduction is also associated with the standard interface modularity. Partitioning code into small functional units, allows individual sections to be used by multiple applications. For example, a company may have several departments that all use a different application for printing invoices. On the bottom of each invoice, the weekday on or closest to 30 days from the print date is given as the last day for returns or warranty claims. When the company decides to increase this to a 60-day period, the maintenance programmer only needs to adjust and recompile the single, common, futureDate class, rather than having to alter and recompile a separate application for each department. The potential savings of such a practice grows with the reuse of classes from other projects. (Post 2002) A review of the section on reuse will show that this argument for OO may not typically be valid, as module reuse is not always practiced.

H. Proper Use of OO Concepts

Another reason OO applications have the potential to yield a lower maintenance cost is based on the proper use of the previously discussed aspects of OO. If these aspects are properly implemented within a company, the resultant code should have a substantially lower bug rate, as most errors will have been trapped in earlier stages of development or in previous application development projects. For any of these aspects to work, a company must take an active stance and demand that employees adhere to a policy of methodical, if not religious, programming techniques. A review of the previous sections will show that proper use of these aspects is infrequently found in practice, thus negating this assertion.

I. Learning Curves, Specialization, and Salary

Object Oriented applications are, on average, more complex and more robust than procedural applications. In order to create more advanced applications, a more advanced and expressive programming language is required. This creates a steeper learning curve, which means object programmers, on average, need more training than procedural programmers. (Fayad & Douglas 1997) Having a more educated programmer is a requirement that also trickles down into the maintenance-programming department. In the past, it was possible for companies to hire new programmers and give them on-the-job training while they maintained older applications. This is not always the case with Object Oriented maintenance. Because of the initially steep learning curve, maintenance programmers are now required to be more educated. Thus, the practice of hiring cheap, unseasoned maintenance programmers has been replaced by a system where the maintenance programmer can command a salary equal to the development programmer's. Adding to this problem, OO application maintenance may take over 100% longer than fixing a fault in a procedural application. (Hatton 1999) With a longer time required for system maintenance and a higher programmer salary, the end result is clear – maintenance costs a lot more for an OO application.

On a related topic, another effect of using a more complex programming language is programmer specialization. In much the same way as a neurosurgeon is paid more than a general practitioner, a specialized OO programmer is paid more than a procedural language programmer. In fact, the average OO programmer makes 20% more money than traditional language programmers. (Feiman & Frey 2000) This means that to save money over traditional language maintenance costs, a minimum of 16% fewer mistakes must be corrected throughout the applications lifetime. Unfortunately, evidence suggests that due to the complexity of Object Oriented applications, the error rate has not decreased [See Complexity] [See Testing]. In fact it has been found that OO applications tend to have a greater fault density than procedural applications and as time progresses the number of discovered faults increase at a higher rate than with procedural applications (in other words, linear regression of faults found by time yields a steeper slope for OO applications). Therefore, maintenance-programming costs are not reduced, yielding no significant cost advantages to Object Oriented applications.

J. Inheritance

The OO paradigm promised a framework that allowed application maintenance to seamlessly integrate new business rules into an existing system; clearly offering an advantage over procedural applications. The integration of new business rules often does not necessitate a large and totally new section of (redundant) code to be written; instead the programmer can inherit the attributes and methods from existing classes and expound on them. Chubb Insurance, for example, found that they were able to implement new business rules six times faster in an OO language than in a procedural language, due in part to inheriting from existing classes (IBM 1997). Inheritance, in theory, provides an easy way to use previously proven code, hence creating an application of higher quality and in less time. While the theory is sound, and success stories do exist (albeit limited), in practice it is found that the number of times previous code is inherited from is only a small percentage. The running joke among programmers is that inheritance offers a programmer the opportunity to inherit the errors of a previously written class. This pervasive mind-set, regardless of accuracy, is a major reason inheritance is not used at the maintenance level [See Extensibility].

K. Conclusion

Clearly one can see that opportunities abound where reduced maintenance costs could turn a predestined procedural code failure into a highly successful OO project. Unfortunately, the benefits OO might provide in the maintenance arena are seldom seen. The reason empirical evidence is hard to come by is three-fold. First, very few OO projects are underway in the business world. Companies are far too afraid to put money into an unproven programming technique, thus propagating an endless loop. Second, too few of the implemented OO projects follow all of the necessary paradigm rules to allow maintenance issues to be studied in detail. Lastly, compared to more traditional programming paradigms, OO systems have not been in place long enough to require any significant maintenance work to be performed. (Card & Emam & Scalzo 2001) Effectively, no clear answer may be asserted which accurately conveys the financial savings potential of long-term OO maintenance.

XI. Conclusion

To what extent, as the title of this report asks, has the object-oriented paradigm kept its promise? The seminar participants are like a microcosm of the technical community on this question. There are enthusiasts, skeptics, and gradations between these extremes. At the end of the semester, the author of each "technical hope" section evaluated how well his topic supported the management hopes identified in the introduction. The Kept Promise scale varies from 1 (not at all) to 5 (completely) with 3 being a neutral value.

	Faster	*Better*	*Cheaper*	*More*	*Average*
Complexity	3	1	2	3	**2.25**
Naturalness	4	3	2	1	**2.5**
Modeling	1	3		3	**2.33**
Extensibility		3		3	**3**
Reuse	4	3	3	4	**3.5**
Easier testing	4	2	1		**2.33**
Maintenance		3	3		**3**

Aristotle cautions us not to demand more precision than a subject supports, so averages to hundredths on an ordinal scale are presumptuous. Still, the combined assessment is hard to ignore: object-oriented technology has only partly kept its promises. One practitioner cautions that object-oriented technology "is not for dummies." Like so many things in life, it takes effort to reap the rewards. In his response to Frederick P. Brooks' famous paper, "No Silver Bullet" (Brooks, 1987), David Harel argues that software developers must bite the bullet (Harel 1998), and use more powerful representations for more difficult software. The promise of the object-oriented paradigm demands proportionate effort from it's would-be benefactors.

XII. Works Cited

A Gentle Introduction to Software Engineering. [Online], Available at: http://stsc.hill.af.mil/resources/tech_docs/entire.asp?name=gise [11 November 2002].

Abdurazik, A. and Offutt, J. (2000), *Using UML collaboration Diagrams for Static Checking and Test Generation*, [Online], Available from http://www.isse.gmu.edu/~ofut/rsrch/papers/uml00.pdf [1 October 2002].

Act World Training, [Online], Available from: http://www.actworld.net/services/training/programming/cobol_inter.asp [2 December 2002].
Available from http://www.math.luc.edu/~glance/Spr02/473/lecture/473-4.htm [28, Oct. 2002].

Beierle, N. (2001), *Object Oriented Software Testing*, [Online], Available from http://www.tncc.vccs.edu/staff/beierle/new_page_5.htm [1 October 2002].

Berard, E V. (2002), *Life-Cycle Approaches*, [Online], Atomic Object, Available from: http://www.toa.com/pub/life_cycle_article.txt [14 October 2002].

Berard, E. V. (2002), *Creation of, and Conversion to, Object-Oriented Requirements*, [Online], Atomic Object, Available from: http://www.toa.com/pub/cc_oor_article.txt [14 October 2002].

Berard, E. V. (2002), *Issues in the Testing of Object-Oriented Software*, [Online], Available from: http://opax.swin.edu.au/~303847/itd511/week3/oo_test.txt [9 September 2002].

Binder, R. V. (14 May 1996), *Testing Object-Oriented Systems: A Status Report*, [Online], Available from http://www.rbsc.com/pages/oostat.html [19 September 2002].

Binder, R. V. (15 October 2001), *Object-Oriented Testing: Myth and Reality*, [Online], Available from http://www.rbsc.com/pages/myths.html [19 September 2002].

Booch, G, (1987), *Software Engineering with Ada,* 2nd edn*,* Benjamin/Cummings, Redwood City.

Booch, G. (1991), *Object-Oriented Design With Applications*. Benjamin Cummings.

Booch, G. 1994, *Object-Oriented Analysis and Design*, 2nd edn, Benjamin/Cummings, Redwood City.

Borstler, J., Johansson, T. & Nordstorm, M. 2002, 'Teaching OO Concepts – A Case Study using CRC-Cards and BlueJ' in 32nd ASEE/IEEE Frontiers in Education Conference, Boston MA, pp1-6.

Bowen, B. D., (May 1997), *Software Reuse With Java^{TM} Technology: Finding The Holy Grail*, [Online], Available from: http://java.sun.com/features/1997/may/reuse.html [9/10/2002]

Brooks Jr., Frederick P.1987, 'No Silver Bullet-Essence and Accidents of Software Engineering, *IEEE Computer*, Vol. 20, No. 4pp. 10 - 19.

Budd, T. (1997), *Object Oriented Programming 2nd Edition*, Addison-Wesley, California.

Card, D.N., Emam, K.E. & Scalzo, B. (2001), Measurement of Object-Oriented Software Development Projects, Software Productivity Consortium NFP [Online] Available at http://sern.ucalgary.ca/courses/SENG/693/F00/ [15 September 2002]

Cardelli, L. (7 December 1999), *Bad Engineering Properties of Object-Oriented Languages*, [Online], Available from http://research.compaq.com/SRC/articles/199702/BadPropertiesOfOO.html [19 September 2002].

Caspersen, M. E, Madsen, O. L. & Skov, S. H. (2001), *Testing Object-Oriented Software,* [Online], Available from http://www.cit.dk/COT/ [8 September 2002].

Coad, P. and Yourdon, E. 1991, *Object-Oriented Analysis*, 2nd edn, Prentice-Hall, Upper Saddle River.

Coatta, T., (July 2000), *To Reuse or Not to Reuse, That is the Question*, [Online], Available from: http://www.spc.ca/essentials/jul1200.htm#3 [2 September 2002].

Cockburn, A., 1998, *Surviving Object-Oriented Projects, A manager's Guide*, Addison Wesley, Longman Inc.

Cockburn, A., 2002, *Writing Effective Use Cases*, Addison-Wesley.

Comp 473 Lecture 4 (2002), *Object Oriented Programming*, [Online],

Cringely, R. X. (1996), *A History Of The Computer,* [Online] http://www.pbs.org/nerds/timeline/pre.htm [17 October 2002].

Dawson, L, and Swatman, P, (1993), "The Use of Object-Oriented Models in Requirements: A Field Study,' *Proceedings of the 20th international conference on Information Systems*, p.260-273, December 12-15, 1999, Charlotte, North Carolina, United States

Dlugosz, J. M. (1994), 'The Dark Side of OOP', American Programmer , Vol.7, no.10.

Eckel, B., *Thinking In Java*, 2nd. ed. Prentice Hall PTR, 2000

Erickson, C. (2001), *Inheritance, Abstraction and Identity*, [Online], Available from: http://www.atomicobject.com/training/training-material/oo-notes/inheritance.html [9/5/2002]

Erickson, C. (30 May 2002), *OO Testing: from academia to the real world*, [Online], Available from http://www.atomicobject.com/download/testingtalk.pdf [19 September 2002].

Fayad, M. and Schmidt, D. (1997), 'Object Oriented Application Frameworks', Communications of the ACM, Vol. 40, no. 10.

Feiman, J. and Frey, N. (2000), Migrating Legacy Developers to JAVA: Costs, Risks and Strategies, Gartner Group.

Fontoura, M. F. (2002), Object Oriented Application Frameworks: the Untold Story, Pontifical Catholic University of Rio de Janeiro, [Online] Available at http://www.almaden.ibm.com/cs/people/fontoura/papers/untold.pdf [17 October 2002]

Fowler, M. (2001) *Is Design Dead,* [Online] Available from http://matinfowler.com/articles/designDead.html [3 October 2002]

Frost, A., *Software Reuse: Concepts and Applications*, [Online], Available from: http://www.cs.uwindsor.ca/users/n/niu1/chp08.html

Gamma, E., Helm, R., Johnson, R., Vlissides, J., (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, New Jersey

Hamza, H. and Fayad, M. E., (2002), *Model-based Software Reuse Using Stable Analysis Patterns*, [Online], Available from: http://research.intershop.com/workshops/ECOOP2002/papers/HamzaFayad/HamzaFayad.pdf [10/5/2002]

Harich, J., (8/1998), Pitfalls of Components, [Online], Available from: http://smile.jcon.org/soft/article/future_app_dev/PitfallsOfComponents.html [9/20/2002]

Harel, David. (1998), 'Biting the Silver Bullet: Toward a Brighter Future for System Development', *IEEEComputer*, Vol. 15, no. 1, pp. 8 - 20..

Hatton, L. (1998), 'Does OO Sync With How We Think?', *IEEE software*, Vol. 15, no. 3, November 1997.

Henderson-Sellers, B. and Edwards, J, (1990) 'The Object-Oriented Systems Life-Cycle', *Communications of the ACM,* Vol. 33, No 9, September 1990, p142-159

Holmevik, J. R. (1995), *The History of Simula,* [Online] http://java.sun.com/people/jag/SimulaHistory.html [2 November 2002].

Hoydalsik, G. M. and Sindre, G, (1993) 'On the purpose of Object-Oriented Analysis' *ACM OOPSLA '93,* pages 240-255.

Hunter, R., (3/1997), *Once is not Enough*, [Online], Available from: http://www.cio.com/archive/030197_gartner.html [9/15/2002]

IBM (1997), Chubb Creates New Client/Server "Masterpiece" with VisualAge, [Online] Available at http://www-3.ibm.com/software/ebusiness/jstart/casestudies/ [17 October 2002]

IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990.

Irwin, G. and Monarchi, D. (1996), *Reuse and Analogical Reasoning in Object-Oriented Analysis*, [Online], Available from: http://hsb.baylor.edu/ramsower/ais.ac.96/papers/irwin.htm [10/2/02]

Jacobs B. (2002), [Online], Available from http://www.geocities.com/tablizer/model.htm, [16 October 2002].

Jacobson, I., Martin, G. & Patrik, J., (1997), *Software Reuse – Architecture, Process and Organization for Business Success*, ACM Press, New York

Java 2 SDK, Standard Edition Documentation, Rev 1.4.1, Sun Microsystems, Santa Clara.

Java Object Serialization Specification, Rev 1.4.4, Sun Microsystems, Santa Clara.

Johnson, R. A. (2000), 'The Ups and Downs of Object-Oriented Systems development', Communications of the ACM, Vol. 43 no. 10

Johnson, R. A., Hardgrave, B. C., Doke, E. R., An industry analysis of developer beliefs about object-oriented systems development, ACM SIGMIS Database, Volume 30, Issue 1, January 1999.

Jorgensen, P. C. 1995, *Software Testing - A Craftsman's Approach*, CRC Press LLC, Boca Raton.

Jorgensen, P. C. 2002, *Why is Testing Object-Oriented Software So Hard?*, presented at Division of Science and Mathematics Seminar.

Jorgensen, P. C. and Erickson, C. 1994, 'Object-oriented integration testing', *Communications of the ACM*, Vol. 37, No. 9, pp.30-38.

Karakap, U., and Sultanoolu, S., (15 October 1998), *Object Oriented Metrics,* [Online], Available from http://yunus.hun.edu.tr/~senser/oom.html [8 September 2002].

Kim, H. and Wu, C. 1996, 'A class testing technique based on data bindings', in *Testing Object Oriented Software*, eds D. Kung, P. Hsia & J. Gao, IEEE Computer Society, Los Alamitos, pp. 58-63.

Lam, Josie (1997), *Object-Oriented Technology*, [Online] http://disc.cba.uh.edu/~rhirsch/spring97/lam1/hope.htm

Loy, P. H., 1990, 'A Comparison of Object-Oriented and Structured Development Methods', *Software Engineering Notes,* Vol. 15, No. 1, January 1990, pages 44-48.

Marick, B., (July 1995), *Notes on Object-Oriented Testing,* [Online], Available from http://www.testing.com/writings/1-fault.htm [8 September 2002].

McClure, C., (1995), *Model-Driven Software Reuse: Practicing Reuse Information Engineering Style*, [Online],Available from: http://www.reusability.com/papers2.html [9/30/2002]

McGregor, J. D, and Korsen, T., (1990) 'Understanding Object-Oriented: A Unifying Paradigm', *Communications of the ACM,* Vol. 33, No 9, September 1990, pages 40-60.

McGregor, J. D., and Korsen, T., (1990) 'Introduction', *Communications of the ACM,* Vol. 33, No 9, September 1990, p38.

Meyer, B. 1988, *Object-Oriented Software Construction*, Prentice-Hall, Upper Saddle River.

Meyer, B. 1997, *Object-Oriented Software Construction*, 2nd edn, Prentice-Hall, Upper Saddle River.

Montlick, T., (1999), *What is Object-Oriented Software?,* [Online], Available from http://softwaredesign.com/objects.html [8 September 2002].

Nader, N. and Rine, D.C., (1998), *A Validated Software Reuse Reference Model Supporting Component-Based Management*, [Online], Available from: http://www.sei.cmu.edu/cbs/icse98/papers/p13.html [28 September 2002].

Nierstrasz, O., (2002), *1. P2 — Object-Oriented Programming*, [Online], Available from http://www.iam.unibe.ch/~scg/Teaching/P2/intro.pdf [15 Oct. 2002].

Nygaard, K. and Dahl, O., (1996), *How Object-Oriented Programming Started,* [Online], Available from: http://www.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html#Stopp

Palmer, R., 2002, *Effective Diagramming Techniques*, Masters Thesis, Grand Valley State University.

Pitfalls in OO Analysis, Atomic Objects, [Online], Available from: http://www.atomicobject.com/training/training-material/oo-notes/oo-analysis-pitfalls.html [11 November 2002].

Post, E. (2002), Advantages of using the Object-Oriented Paradigm for Designing and Developing Software, [Online] Available at http://www.lincoln.ac.nz/amac/profiles/poste.htm [29 October 2002]

Poulin, J.M. and Werkman, K. J., (1994), *Software Reuse Libraries with Mosaic*, [Online], Available from: http://archive.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/werkman/www94.html [9/30/2002]

Pressman, R.1997, *Software Engineering: A Practioner's Approach*, 4th edn, McGraw-Hill, New York

Rine, D. C., (1997), *Success factors for software reuse that are applicable across domains and businesses*, [Online],Available from: http://doi.acm.org/10.1145/331697.331736

Schmidt, D. C., (10/1996), *Software Patterns*, [Online], Available from: http://www.cs.wustl.edu/~schmidt/CACM-editorial.html [9/30/2002]

Schmidt, D. C., (January 1999), *Why Software Reuse has Failed and How to Make It Work For You*, [Online], Available from: http://www.cs.wustl.edu/~schmidt/reuse-lessons.html [9/30/2002]

Sommerville, Ian (1992), *Software Engineering 4th Edition*, Addison-Wesley, England.

Strong, N. S., 1992, 'Identifying a Complete Object Oriented Life Cycle for Large Systems Development', *ACM,* January 1992, pages 166-175.

Stroustrup, B. 1991, *What is object-oriented programming?* [Online], Available at: http://www.research.att.com/resources/articles/whatis.pdf [28 October 2002].

Sultanoolu, S. and Karakab, U., (10/1998), *Object Oriented Metrics*, [Online], Available from: http://yunus.hun.edu.tr/~sencer/oom.html [9/9/2002]

Taylor, D. 1990, *Object-Oriented Technology: A Manager's Guide*, Servio, Alameda.

Tsi, Wei-Tek. (2002), *Testing Object-Oriented Software,* [Online], Available from http://asusrl.eas.asu.edu/est/content/oo/oo.pdf [8 September 2002].

Wieringa, R., 1998. 'A Survey of Structured and Object-Oriented Software Specifications Methods and Techniques', *ACM Computing Surveys*, Vol. 30, No. 4 December 1998, pages 459-527.

Williamson, M., (March 1997a), *Cultural Issues*, [Online], Available from: http://www.cio.com/archive/030197_cultural.html [20 September 2002].

Williamson, M., (March 1997b), *Introduction*, [Online], Available from: http://www.cio.com/archive/030197_intro.html [20 September 2002].

Williamson, M., (March 1997b), *Technology*, [Online], Available from: http://www.cio.com/archive/030197_technology.html [20 September 2002].

Winter, M., (1998), *Managing Object-Oriented Integration and Regression Testing (without becoming drowned),* [Online], Available from http://www.informatik.fernuni-hagen.de/import/pi3/publikationen/abstracts/EuroSTAR98.html [2 November 2002].

Wirfs-Brock, R., Wilkerson, B. & Weiner L. 1990, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs.

York, R., *COBOL Training*, [Online], Available from: http://www.coboltraining.co.uk/coboltraing/details.com [2 December 2002].

Yourdon, E., *The Year of the Object,* [Online] *http://www.yourdon.com/articles/8908clm.html*

Appendix A – Author Information

Dr. Paul C. Jorgensen

Paul gradated with a B.A. in Mathematics from North Central College in 1964, an M. A. in Mathematics from the University of Ilinois in 1965, and a Ph.D. in Computer Science from Arizona State University in 1985. He worked in various divisions of GTE Corp. 20 years, and then joined the faculty of Arizona State University for two years. He has been at Grand Valley State University since 1988, and is a full professor in the Computer Science and Information Systems department.

His teaching, research, and consulting interests coincide on models for requirements specification and software testing. He is the co-author of two books and the sole author of Software Testing-A Craftsman's Approach, 1st and 2nd editions. He is a Senior Member of IEEE Computer Society.

David Fernandez

Graduated from Grand Valley State University in 1993 with a Bachelor of Business Administration (Finance & Accounting). Worked one year with ACEMCO of Grand Haven, developing their Activity-Based Cost system infrastructure. Joined Steelcase, Inc. in January of 1995, serving as Manufacturing Financial Analyst and SAP systems integrator for the Wood division facilities. Developed real-time work-in-process inventory tracking systems and lead the conversion from standards based systems to activity based tracking systems, using DB2 and Visual Basic interfaces, eventually leading to Master Planner integration. Developed and deployed (1999-2000) a distributed (client-server desktop app) expense tracking and budgeting system using Sql Server, Visual Basic and Java interfaces to all North American offices. In August of 2000, joined the Services division of Steelcase as a Senior Applications Engineer (lead developer), designing and implementing N-Tier web applications (Order Fulfill, CRM, Project Mgmt, etc) utilizing J2EE and .NET (ASP/WebServices/Remoting; C#/VB) with MS Sql Server 2000 and Oracle 8-9i supporting data tiers.

Al Fischer

Al graduated from Calvin College in 1988 with a degree in Mathematics and Computer Science. He spent the next two years teaching math and computer courses at East Grand Rapids High School. In 1990, he joined EDS as part of their Systems Engineer Development program. After completion of the program, he worked in various installations for EDS serving their largest customer, GM. After seven years at EDS, Al joined Amway as Programmer Analyst. In 2002, he became a part of the DBA team, where he still works. Al has been married to Melanie since 1988 and has two sons, Avery and Clark.

Michael Greco

B.S. Major in Computer Science/Mathematics. Minor in Physics. M.S. Concentration in Object Oriented Programming/Software Engineering. 7 years in software development, primarily Java and C programming.

Bradly M. Hussey

Bradly M. Hussey graduated from Grand Valley State University in 2000 with a Bachelors of Science degree. Brad will graduate from Grand Valley State University with a Master's Degree in April, 2003.

Steven Kuchta

Graduated Grand Valley State Colleges-Thomas Jefferson College 1975 with a Bachelor of Philosophy.

Graduated GVSU 1988 with a Bachelor of Science, emphasis in Computer Science.
I've been working in the field since graduation. I have held a number of positions and worked with several different environments.

I am, most recently sub-contracted to a defense contractor and working in a Unix/Linux environment designing and implementing Java Client/Server solutions and working on software process improvement projects.

I am hoping to complete my Master's level work in Computer Science in Dec. 2002.

Hong Li

B.S. Major in Engineering in China.
M.S. Computer Information System.

Steven Overkamp

Steve graduated from Michigan Technological University in 1974 with a degree in Electrical Engineering. After working for three years designing electrical systems for industrial manufacturing facilities for the Austin Company in Des Plaines, IL., Steve then worked for Consumers Power Company at various locations around the state of Michigan. In 1996, Steve graduated from Saginaw Valley State University with a degree in Computer Information Systems. Steve joined Keane, Inc as a Consultant in 1998, where he still works. He is presently maintaining several web based applications for Pharmacia in Kalamazoo, MI.

Douglas Rodenberger

Graduated from Northern Michigan University in 1987 with a Bachelor of Business Administration, Computer Information Systems. Worked for 7 years at Ace Hardware Corporation in Oak Brook, IL as a systems programmer and analyst working with electronic publishing systems, freight traffic systems, promotional planning systems and database design for end-user querying and reporting. Currently working for Novartis Consumer Health in Fremont, MI as a Sr. Analyst/Programmer-Datamart Architect working with data warehouse design and construction and e-business projects utilizing J2EE client server technology.

Richard VanderWal

Rick VanderWal is currently a Software Engineer for Smiths Aerospace developing objected-oriented applications using C++. Areas of specialization include Simulation, Networking, and Real Time Operating Systems application development. He graduated from the United States Military Academy in 1989 with Bachelor of Science degree in Computer Engineering.